目錄

操作系统的基本原理与简单实现

--基于*ucore OS + RISC-V*

对于在校的学生和已经参加工作的工程师而言，能否以较小的时间和精力比较全面地了解操作系统呢？陆游老夫子说过“纸上得来终觉浅，绝知此事要躬行”，也许在了解基本的操作系统概念和原理基础上，通过实际动手来一步一步分析、设计和实现一个微型化的操作系统，会发现操作系统原来如此，概念原理和实际实现之间有紧密的联系和巨大的差异。

早期开放开源的UNIX操作系统和MIT教授 Frans Kaashoek 等基于UNIX v6设计的xv6操作系统给了我们启发：对一个计算机专业的本科生而言，设计实现一个操作系统有挑战但是可行！但x86相对封闭&复杂和有一定历史包袱的CPU硬件接口给OS学习带来了一定的挑战。1980年前后，UC Berkeley的Dave Patterson主导了Berkeley RISC项目并设计了其第一代的处理器RISC I，并在2014年发展到了开放&开源的第五代指令集架构RISC-V。本书想进行这样的教学尝试，以操作系统基本原理为教学引导，以简洁的RISC-V CPU为底层硬件基础，设计并实现一个微型但全面的“麻雀”操作系统—ucore。期望能够采用简化的计算机硬件为基础，以操作系统的基本概念和核心原理为实践指导，逐步解析操作系统各种知识点和对应的实验，做到有“理”可循和有“码”可查，最终让读者了解和掌握操作系统的原理、设计与实现。

陈渝 向勇

清华大学计算机系

2018年春

操作系统的基本原理与简单实现

--基于*ucore OS + RISC-V*

- 陈渝 yuchen AT tsinghua.edu.cn
- 向勇 xyong AT tsinghua.edu.cn

2018年

简介

用一句话描述本章

站在一万米的高空看操作系统的发展和特征！

本章概述

本章站在一万米的高空来看操作系统和计算机原理。相对用软件而言，操作系统其实是一个相比较复杂的系统软件，直接管理计算机硬件和各种外设，以及给应用软件提供帮助。这样描述还太简单了一些，我们可对其进一步描述：操作系统是一个可以管理CPU、内存和各种外设，并管理和服务应用软件的软件。为了完成这些工作，操作系统需要知道如何与硬件打交道，如何更好地面向应用软件做好服务。

本章将讲述操作系统学习的一些基础知识，以及对用于本书的ucore教学操作系统做一个介绍。然后再简单介绍操作系统的基本概念、操作系统抽象以及操作系统的特征。最后还将简要介绍操作系统的历史和基本架构。

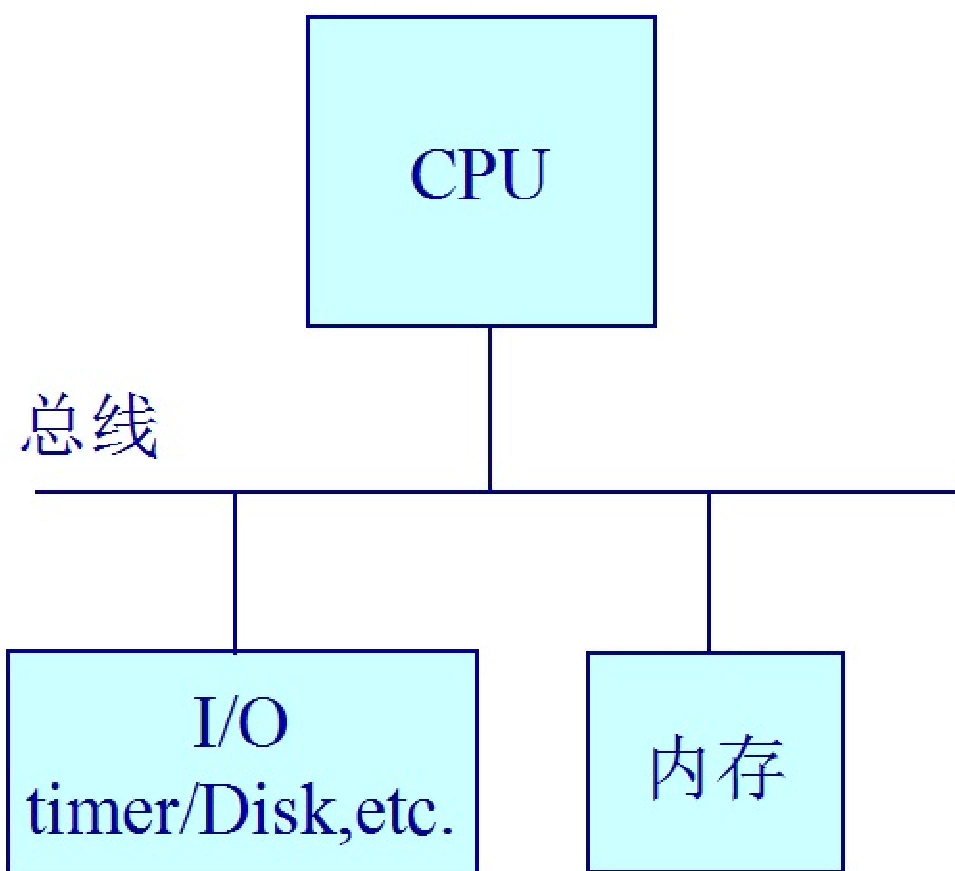
本书希望通过设计实现操作系统来更好地理解操作系统原理和概念。设计实现操作系统其实就是设计实现一个可以管理CPU、内存和各种外设，并管理和服务应用软件的系统软件。为此还是需要先了解一些基本的计算机原理和编程的知识。本书的例子和描述需要读者学习过计算机原理课程、程序设计课程，掌握C语言编程（了解指针等的编程）。如需完成基于RISC-V的ucore实验，则对基于RISC-V的体系结构有一定的了解，大致了解RISC-V的汇编语言。

了解计算机硬件架构

操作系统作为软件，首先要能在计算机硬件上运行，并完成其第一要务：对硬件进行管理和控制。要想深入理解操作系统，就需要了解支撑操作系统运行的硬件环境，即了解处理器体系结构和机器指令集，来探索CPU对操作系统的影响，以及操作系统如何通过各种操作来管理硬件的。当前的硬件和CPU有很多种，在细节上有很大的不同，如果一开始就陷入硬件细节，不利于同学对OS和计算机硬件之间的关系建立一个比较全面和概览性的理解。所以，我们将先简单介绍一个抽象的简化计算机系统，然后在逐步进入到某一具体CPU，即RISC-V，的硬件细节中。

一般计算机硬件架构

这里介绍一下运行操作系统的基本计算机硬件架构。一台计算机可抽象一台以图灵机（Turing Machine）为理想模型，以冯诺依曼架构（ Von Neumann Architecture）为实现模型的电子设备，包括CPU、memory和 I/O 设备。CPU(中央处理器，也称处理器) 执行操作系统中的指令，完成相关计算和读写内存，物理内存保存了操作系统中的指令和需要处理的数据，外部设备用于实现操作系统的输入（键盘、硬盘），输出（显示器、并口、串口），计时（时钟）永久存储（硬盘）。操作系统除了能在计算机上运行外，还要管好计算机。下面将简要介绍计算机硬件以及操作系统大致要完成的事情。



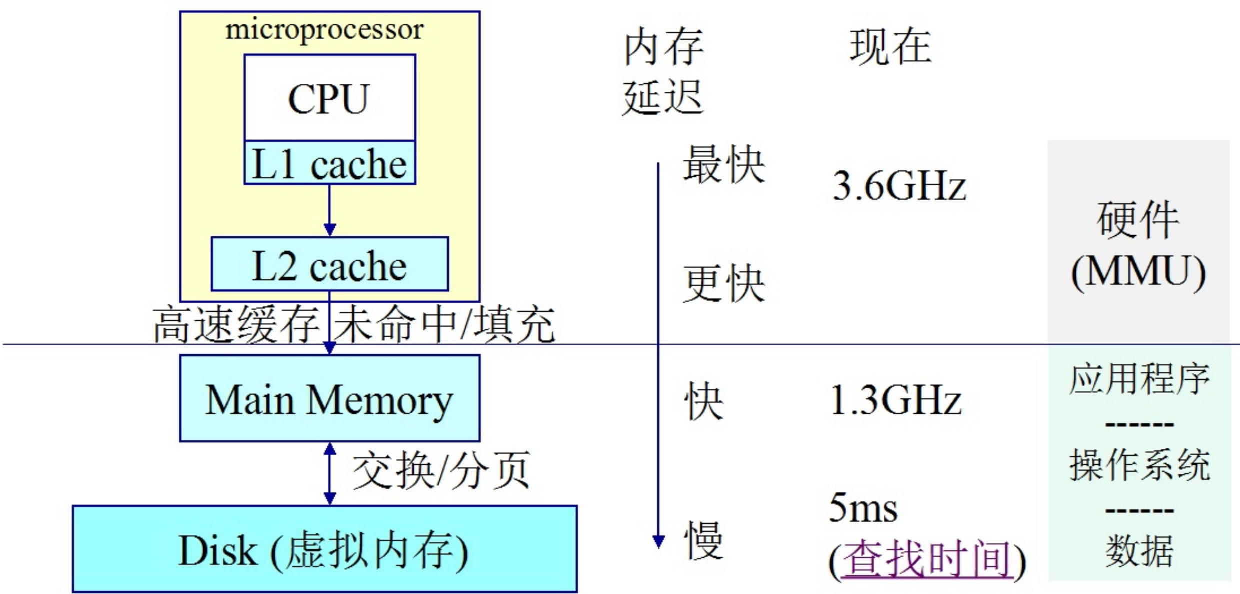
CPU

CPU是计算机系统的核心，目前存在各种8位，16位，32位，64位的CPU，应用在从嵌入式系统到巨型机等不同的场合中。CPU从一加电开始，从某设定好的内存地址开始，取指令，执行指令，并周而复始地运行。取指令的过程即从某寄存器（比如，程序计数器）中获取一

个内存地址，从这个内存地址中读入指令，执行机器指令，不断重复，CPU运行期间会有分支和调用指令来修改程序计数器，实现地址跳转，否则程序计数器就自动加１，让CPU从下一个内存地址单元取指令，并继续执行。

由于CPU执行速度很快（x86 CPU可达到2GHZ以上的时钟频率，RISC-V CPU可达到1.5 GHZ的时钟频率），如果当前可以运行的程序太少，则会出现CPU无事可做的情况，导致计算机系统效率太低。这时就需要操作系统来帮忙了，我们需要操作系统除了能管理硬件外，还能管理应用程序，让它们能够按一定的顺序和优先级来依次使用CPU，充分发挥CPU的效能。操作系统管一个程序比较容易，但如果管理多个程序的运行，就需要考虑如何分配CPU资源的问题，如何避免程序执行期间发生“冲突”的问题等，这是操作系统需要完成的重要功能之一。

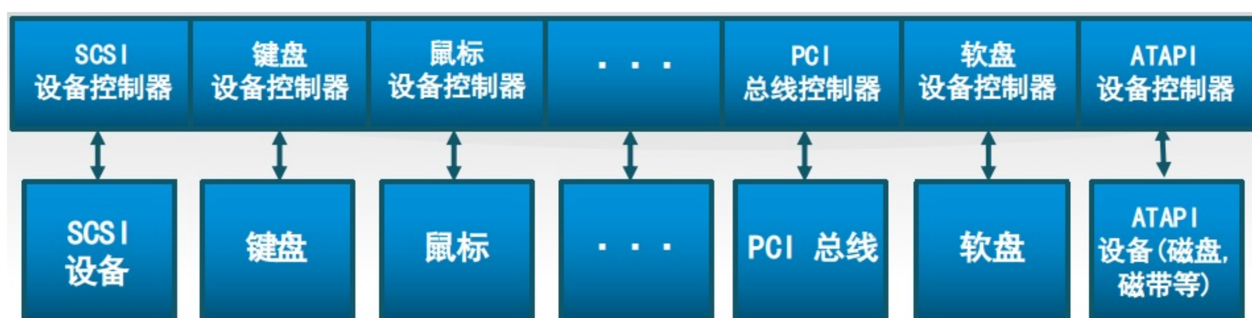
memory



计算机中有多种多层次的存放数据和指令代码的硬件存储单元，比如在CPU内的寄存器（register）、高速缓存(cache)、内存（memory）、硬盘、磁带等。寄存器位于CPU内部，其访问速度最快但成本昂贵，在对于传统的CISC（复杂指令集计算机，如Intel 80386处理器）中一般只有几个到十个左右的通用寄存器，而对于RISC（精简指令集计算机，如RISC-V），则可能有几十个以上通用寄存器；高速缓存（cache）一般也在CPU内部，cache是内存和寄存器在速度和大小上的折衷，比寄存器慢2~10倍，容量也有限，量级大约几百KB到几十MB不等；再接下来就是内存了，内存位于CPU外，比寄存器慢10倍以上，但容量大，目前一般以几百兆B到几百GB不等；硬盘容量更大，但一般比寄存器要慢1000倍以上，不过掉电后其存储的数据不会丢失。

由于寄存器、cache、内存、硬盘在读写速度和容量上的巨大差异，所以需要操作系统来协调数据的访问，尽量主动协助应用软件，把最近访问的数据放到寄存器或cache中（实际上操作系统不能直接控制cache的读写），把经常访问的数据放在内存中，把不常用的数据放到硬盘上，这样可以达到让多个运行的应用程序“感觉”到它可用使用很大的空间，也可有很快的访问速度。如何让在运行中的每个程序都能够得到“足够大”的内存空间，且程序间相互不能破坏对方的内存“领地”，且建立他们之间的“数据共享”通道，这是操作系统需要完成的重要功能之一。

I/O



CPU处理的数据需要有来源和输出，这里的来源和输出就是各种外设，如大家日常使用计算机用到的键盘、鼠标、显示器、声卡、GPU、U盘、硬盘、SSD存储、打印机、网卡、摄像头等。上图中给出了PC计算机的一些外设硬件。应用程序如果直接访问外设，会有代码实现复杂，可移植性差，无法有效并发等问题，所以操作系统给应用程序提供了简单的访问接口，让应用程序不需要了解硬件细节。具体访问外设的苦活累活都交给操作系统来完成了，这就是操作系统中外设驱动哦程序的主要功能。

如果操作系统要通过CPU对数据进行加工，首先需要有输入，在处理完后还要进行输出，否则没东西要处理或者执行完了无法把结果反馈给用户。操作系统要处理的数据需要从外设（比如键盘、串口、硬盘、网卡）中获得，这就是一种读外设数据的操作；且在处理完毕后要传给外设（比如显示器、硬盘、打印机、网卡）进一步处理，这其实就是一种写外设数据的操作。

一般而言，IO外设把它的访问接口映射为一块内存区域，操作系统通过来用通常的内存读写指令来管理设备；或者CPU提供了特定的IO操作指令，操作系统通过这些特定的指令来完成对IO外设的访问。并且操作系统可以通过轮循、中断、DMA等访问方式来高效地管理外设。

比如，在RISC-V中通过对特定地址的内存（代表IO外设的接口）进行读或写访问，就可以实现对IO外设的访问了。在Intel x86中有两条特殊的 `in` 和 `out` 指令来在完成CPU对外设地址空间的访问，实现对外设的管理控制。本书不会涉及很多复杂具体硬件，而只涉及到操作系统用到的一些最基本的外设硬件（时钟，串口，硬盘）细节。

RISC-V硬件架构

通用CPU一般能够在硬件上支持内存空间的隔离，使得多个程序在各自独立的内存空间中并发执行。这种硬件机制即支持用户特权级和内核特权级。应用程序运行在用户特权级，这样应用不能执行特权指令，且不能破坏操作系统内核的数据和操作系统执行过程。而操作系统内核运行在内核特权级，可以访问特权指令，并管理和控制应用程序，硬件外设等。所以对于操作系统而言，需要CPU硬件至少支持用户特权级和内核特权级（控制隔离），以及内存空间隔离（数据隔离）。

RISC-V是发源于Berkeley的开源instruction set architecture (ISA)。在继续阅读前，对RISCV指令集和架构特别感兴趣的同学可仔细阅读RISC-V的Specifications。当前用户态指令集规范的版本为2.2，特权态指令集规范的版本为1.10。

- User-Level ISA Specification v2.2
- Draft Privileged ISA Specification v1.10

Modular ISA

RISC-V ISA是模块化的，它由一个基本指令集和一些扩展指令集组成

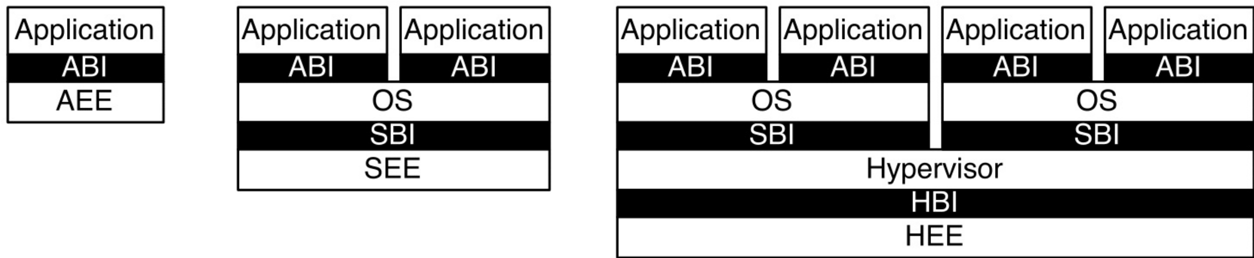
- Base integer ISAs
  - RV32I
  - RV64I
  - RV128I
- Standard extensions
  - M: Integer **M**ultiply
  - A: **A**tomic Memory Operations
  - F: Single-percison **F**loating-point
  - D: **D**ouble-precision Floating-point
  - G: IMAFD, **G**eneral Purpose ISA

举例来说， `RV32IMA` 表示支持基本整数操作和原子操作的32位RISC-V指令集。对于一般在用户态执行的应用程序，只需要上述用户态指令集就基本足够了。但如果是象操作系统这样的软件，需要控制整个计算机硬件，则只使用用户态指令集是不够的，还需要在处于内核特权级的CPU状态下执行特权指令集。

Privileged ISA

Software Stacks

RISC-V在设计时就考虑了硬件虚拟化的需求，三种典型RISC-V系统的结构如下



上图中各个英文缩写对应的全称如下

- ABI: Application Binary Interface
- AEE: Application Execution Environment
- SBI: Supervisor Binary Interface
- SEE: Supervisor Execution Environment
- HBI: Hypervisor Binary Interface
- HEE: Hypervisor Execution Environment

RISC-V通过各层之间的Binary Interface实现了对下一层的抽象，方便了虚拟机的实现以及OS在不同RISC-V架构间的移植。采用了图中第二种结构，bbl在其中充当了SEE的角色。

Privilege Levels

RISC-V共有4种不同的特权级，与x86不同的是，RISC-V中特权级对应数字越小，权限越低

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Hypervisor	H
3	11	Machine	M

一个RISC-V的实现并不要求同时支持这四种特权级，可接受的特权级组合如下

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems
4	M, H, S, U	Systems running Type-1 hypervisors

目前官方的Spike模拟器只部分实现了3个特权级。

Control and Status Registers

RISC-V中各个特权级都有单独的Control and Status Registers (CSRs)，其中应当注意的有以下几个

Name	Description
sstatus	Supervisor status register
sie	Supervisor interrupt-enable register
stvec	Supervisor trap handler base address
sscratch	Scratch register for supervisor trap handlers
sepc	Supervisor exception program counter
scause	Supervisor trap cause
sbadaddr	Supervisor bad address
sip	Supervisor interrupt pending
sptbr	Page-table base register
mstatus	Machine status register
medeleg	Machine exception delegation register
mideleg	Machine interrupt delegation register
mie	Machine interrupt-enable register
mtvec	Machine trap-handler base address
mscratch	Scratch register for machine trap handlers
mepc	Machine exception program counter
mcause	Machine trap cause
mbadaddr	Machine bad address
mip	Machine interrupt pending

在继续阅读前，读者应当查阅Privileged Spec 1.9.1以熟悉以上CSR的功能和用途。

CSR Instructions

RISC-V ISA中提供了一些修改CSR的原子操作，下面介绍之后常用到的 `csrrw` 指令

```
# Atomic Read & Write Bit
cssrw rd, csr, rs
```

语义上等价的C++函数如下

```
void cssrw(unsigned int& rd, unsigned int& csr, unsigned int& rs) {
  unsigned int tmp = rs;
  rd = csr;
  csr = tmp;
}
```

几种有趣的用法如下

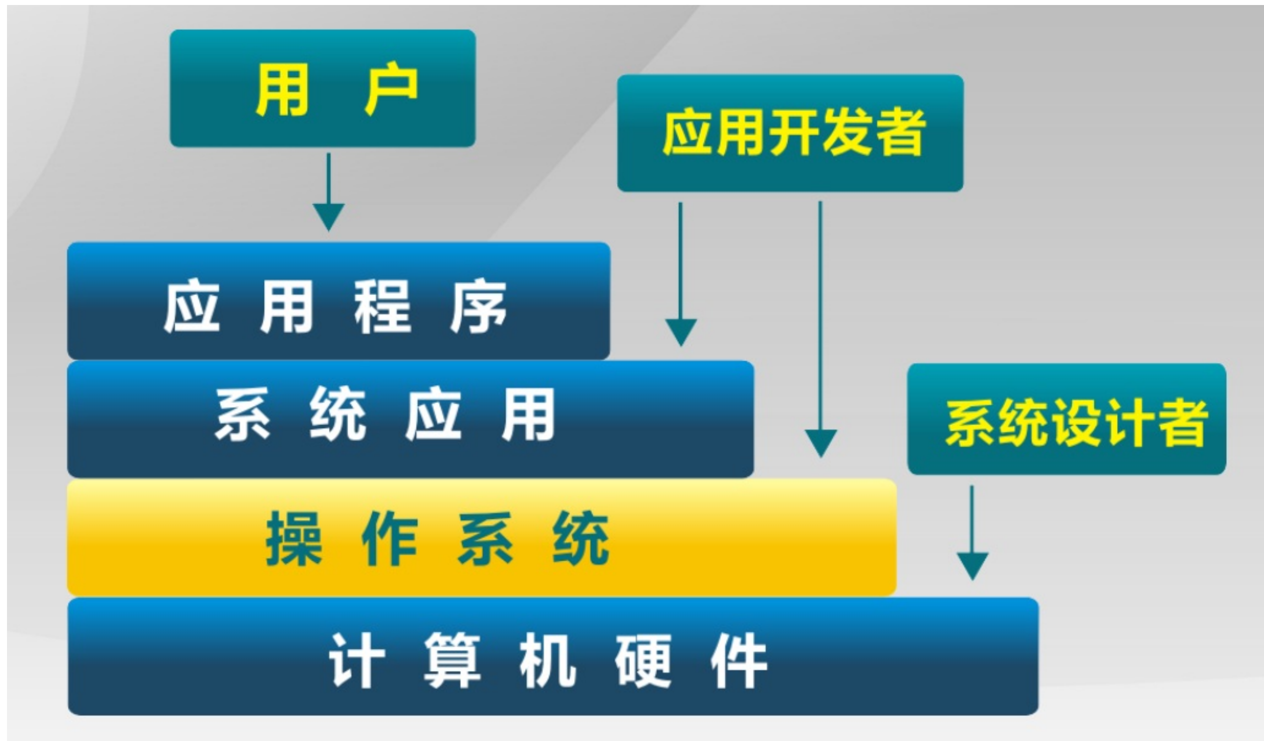
```
# csr = rs
cssrw x0, csr, rs

# csr = 0
cssrw x0, csr, x0

# rd = csr, csr = 0
cssrw rd, csr, x0

# swap rd and csr
cssrw rd, csr, rd
```

我们可以把软件分成应用软件和系统软件，而对于系统软件，我们又可以分为系统应用和操作系统。操作系统是一个大型复杂软件，但如果从使用和实现的抽象角度来看，还是可以用一些比较简洁的描述对其定义、特征等进行归纳总结，让读者有一个比较宽泛的总体理解。



在大众的眼中，操作系统就是他们的手机/终端上的软件系统，包括各种应用程序集合，但在历史上，操作系统也是从无到有地逐步发展起来的。操作系统主要完成对硬件控制和对应用程序的服务所必需的功能，操作系统的历史与计算机发展的历史密不可分。操作系统的内涵和功能随着历史的发展也在一直变化，改进中，在今天，没有图形界面和各种文件浏览器已经不能称为一个操作系统了。

三叶虫时代

计算机在最开始出现的时候是没有操作系统的。启动，扳开关，装卡片/纸带等比较辛苦的工作都是计算机操作员（Operator）或者用户自己完成。操作员/用户带着记录有程序和数据的卡片(punch card)或打孔纸带去操作机器。装好卡片/纸带后，启动卡片/纸带阅读器，让计算机把程序和数据读入计算机机的内存中后，计算机就开始工作，并把结果也输出到卡片/纸带或显示屏上，最后程序停止。

由于人的操作效率太低，计算机的机时宝贵，所以就引入监控程序（Monitor）辅助完成输入，输出，加载，运行程序等工作，这是现代操作系统的起源。一般情况下，计算机每次只能执行一个任务，CPU大部分时间都在等待人的缓慢操作。

恐龙时代

早期的操作系统非常多样化，专用化，生产商生产出针对各自硬件的专用操作系统，大部分用汇编语言编写，这导致操作系统的进化比较缓慢，但进化再持续。在1964年，IBM公司开发了面向System/360系列机器的统一可兼容的操作系统——OS/360。OS/360是一种批处理操作系统。为了能充分地利用计算机系统，应尽量使该系统连续运行，减少空闲时间，所以批处理操作系统把一批作业（古老的术语，可理解为现在的程序）以脱机方式输入到磁带上，并使这批作业能一个接一个地连续处理：1）将磁带上的一个作业装入内存；2）并把运行控制权交给该作业；3）当该作业处理完成后，把控制权交还给操作系统；4）重复1-3的步骤。

批处理操作系统分为单道批处理系统和多道批处理系统。单道批处理操作系统只能管理内存中的一个（道）作业，无法充分利用计算机系统中的所有资源，致使系统整体性能较差。多道批处理操作系统能管理内存中的多个（道）作业，可比较充分地利用计算机系统中的所有资源，提升系统整体性能。二者的共同特点是人机交互性差，这对修改和调试程序很不方便。

爬行动物时代

20世纪60年代末，提高人机交互方式的分时操作系统越来越展露头角。分时是指多个用户和多个程序以很小的时间间隔来共享使用同一台计算机上的CPU和其他硬件/软件资源。1964年由贝尔实验室、麻省理工学院及美国通用电气公司所共同参与研发目标远大的MULTICS(MULTiplexed Information and Computing System)操作系统，MULTICS是一套安装在大型主机上多人多任务的操作系统。 MULTICS以兼容分时系统（CTSS）做基础，建置在美国通用电力公司的大型机GE-645，目标是连接1000部终端机，支持300的用户同时上线。因MULTICS项目的工作进度过于缓慢，1969年AT&T的 Bell 实验室从MULTICS 研发中撤出。但贝尔实验室的两位软件工程师 Thompson 与 Ritchie借鉴了一些重要的Multics理念，以C语言为基础，发展出UNIX操作操作系统。UNIX操作系统的早期版本是完全免费的，可以轻易获得并随意修改，所以它得到了广泛的接受。后来，它成为开发小型机操作系统的起点。由于早期的广泛应用，它已经成为的分时操作系统的典范。

哺乳动物时代

20世纪70年代，微型处理器的发展使计算机的应用普及至中小企及个人爱好者，推动了个人计算机(Personal Computer)的发展，也进一步推动了面向个人使用的操作系统的出现。其代表是由微软公司中在20世纪80年代为个人计算机开发的DOS/Windows操作系统，其特点是简单易用，特别是基于Windwos操作系统的GUI界面，极大地简化了一般用户使用计算机的难度，使得计算机得到了快速的普及。这里需要注意的是，第一个带GUI界面的个人计算机原型起源于伟大却又让人扼腕叹息的施乐帕洛阿图研究中心PARC（Palo Alto Research Center），PARC研发出的带有图标、弹出式菜单和重叠窗口的GUI（Graphical User Interface），可利用鼠标的点击动作来进行操控，这是当今我们所使用的GUI系统的基础。

智人时代

21世纪以来，Housing Internt和移动互联网的迅猛发展，使得在服务器领域和个人终端的应用与需求大增。iOS和Android操作系统是21世纪个人终端操作系统的代表，Linux在巨型机到数据中心服务器操作系统中占据了统治地位。以Android系统为例，Android一词英文本义指“机器人”，它是由Google公司于2007年11月推出的基于Linux Kernel的开源手机操作系统，目前在移动终端中占有最大的份额。Android操作系统是一个包括Linux操作系统内核、基于Java的中间件、用户界面和关键应用软件的移动设备软件栈集合。这里介绍一下广泛用在服务器领域和个人终端中的操作系统内核--Linux操作系统内核。1991年8 月，芬兰学生 Linus Torvalds(林纳斯·托瓦兹)在 comp.os.minix 新闻组贴上了以下这段话：

```
 ”你好，所有使用 minix 的人 -我正在为 386 ( 486 ) AT 做一个免费的操作系统 ( 只是为了爱好 )，不会像 GNU 那样很大很专业。"
```

而他所说的”爱好"就变成我们今天知道的 Linux操作系统内核。 Linus通过Internet首次发表Linux kernel的源代码，并且选用GPL版权协议来发行。GPL版权协议允许任何人以任何形式发布 Linux 的源代码，在Internet的日渐盛行以及 Linux 开放自由的GPL版权之下，吸引了无数计算机Hacker和公司投入开发、改善 Linux kernel，使得 Linux kernel的功能日见强大。

神人时代

当前，大数据、人工智能、机器学习、高速网络、AR/VR对操作系统等系统软件带来了新的挑战。如何有效支持和利用这些技术是未来操作系统的方向。

操作系统的定义与目标

操作系统的定义

有了对硬件的进一步了解，我们就可以给操作系统下一个更准确一些的定义。操作系统是计算机系统机构中的一个系统软件，它的职能主要有两个：对下面（也就是计算机硬件），有效地组织和管理计算机系统中的硬件资源（包括处理器、内存、硬盘、显示器、键盘、鼠标等各种外设）；对上面（应用程序或用户），提供简洁的服务功能接口，屏蔽硬件管理带来的差异性和复杂性，使得应用程序和用户能够灵活、方便、有效地使用计算机。为了完成这两个职能，操作系统需要起到资源管理器的作用，能在其内部实现中安全，合理地组织，分配，使用与处理计算机中的软硬件资源，使整个计算机系统能高效可靠地运行。

操作系统的目标

根据前面的介绍，我们可以看出操作系统有如下一些目标：

- 建立抽象，让上层软件和用户更方便使用；
- 管理软硬件资源，确保计算机系统安全可靠、高性能；
- 其他需求：节能、易用、可移植、实时等等

操作系统的接口

首先，读者可站在使用操作系统的角度来看操作系统。操作系统内核是一个需要提供各种服务的软件，其服务对象是应用程序，而用户（这里可以理解为一般使用计算机的人）是通过应用程序的服务间接获得操作系统的服务的），所以操作系统内核藏在一般用户看不到的地方。但应用程序需要访问操作系统，获得操作系统的服务，这就需要通过操作系统的接口才能完成。如果把操作系统看成是一个函数库，那么其接口就是函数名称和它的参数。但操作系统不是简单的一个函数库，它的接口需要考虑安全因素，使得应用软件不能直接读写操作系统内部函数的地址地址空间，为此，操作系统设计了一个安全可靠的接口，我们称为系统调用接口（System Call Interface），应用程序可以通过系统调用接口请求获得操作系统的服务，但不能直接调用操作系统的函数和全局变量；操作系统提供完服务后，返回应用程序继续执行。

对于实际操作系统而言，具有大量的服务接口，比如Linux有上百个系统调用接口。为了简单起见，以ucore OS为例，可以看到它为应用程序提供了如下一些接口：

- 进程管理：复制创建--fork、退出--exit、执行--exec、...
- 同步互斥的并发控制：信号量--semaphore、管程--monitor、条件变量--condition variable 、...
- 进程间通信：管道--pipe、信号--signal、事件--event、邮箱--mailbox、共享内存--shared mem、...
- 文件I/O操作：读--read、写--write、打开--open、关闭--close、...
- 外设I/O操作：外设包括键盘、显示器、串口、磁盘、时钟、...，但接口是直接采用了文件I/O操作的系统调用接口

> 这在某种程度上说明了文件是外设的一种抽象。在UNIX中（ucore是模仿UNIX），大部分外设都可以以文件的形式来访问

有了这些接口，简单的应用程序就不用考虑底层硬件细节，可以在操作系统的服务支持和管理下简洁地完成其应用功能了。


应用程序
命令行程序
编译器 解释器
系统库
系统调用接口
内核
信号
字符设备I/O
串口驱动
文件管理系统
块设备I/O
磁盘驱动
CPU 调度
虚拟内存管理
物理内存管理
硬件抽象层
串口控制器
终端设备
块设备控制器
磁盘和磁带
存储控制器
物理内存

操作系统抽象

接下来读者可站在操作系统实现的角度来看操作系统。操作系统为了能够更好地管理计算机系统并对应用程序提供便捷的服务，在操作系统的发展过程中，计算机科学家提出了如下四个个抽象概念，奠定了操作系统内核设计与实现的基础。操作系统原理中的其他基本概念基本上都基于上述这四个操作系统抽象。

中断（Interrupt）

简单地说，中断是处理器在执行过程中的突变，用来响应处理器状态中的特殊变化。比如当应用程序正在执行时，产生了时钟外设中断，导致操作系统打断当前应用程序的执行，转而去处理时钟外设中断，处理完毕后，再回到应用程序被打断的地方继续执行。在操作系统中，有三类中断：外设中断（Device Interrupt）、陷阱中断（Trap Interrupt）和故障中断（Fault Interrupt，也称为exception，异常）。外设中断由外部设备引起的外部I/O事件如时钟中断、控制台中断等。外设中断是异步产生的，与处理器的执行无关。故障中断是在处理器执行指令期间检测到不正常的或非法的内部事件（如除零错、地址访问越界）。陷阱中断是在程序中使用请求操作系统服务的系统调用而引发的有意事件。在后面的叙述中，如果没有特别指出，我们将用简称中断、陷阱、故障来区分这三种特殊的中断事件，在不需要区分的地方，统一用中断表示。

进程（Process）

简单地说，进程是一个正在运行的程序。在计算机系统中，我们可以“同时”运行多个程序，这个“同时”，其实是操作系统给用户造成的一个“幻觉”。大家知道，处理器是计算机系统中的硬件资源。为了提高处理器的利用率，操作系统采用了多道程序技术。如果一个程序因某个事件而不能运行下去时，就把处理器占用权转交给另一个可运行程序。为了刻画多道程序的并发执行的过程，就要引入进程的概念。从操作系统原理上看，一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。操作系统中的进程管理需要协调多道程序之间的关系，解决对处理器分配调度策略、分配实施和回收等问题，从而使得处理器资源得到最充分的利用。

虚存（Virtual Memory）

简单地说，虚存就是操作系统通过处理器中的MMU硬件的支持而给应用程序和用户提供一个大的（超过计算机中的内存条容量）、一致的（连续的地址空间）、私有的（其他应用程序无法破坏）的存储空间。这需要操作系统将内存和硬盘结合起来管理，为用户提供一个容量

比实际内存大得多的虚拟存储器，并且需要操作系统为应用程序分配内存空间，使用户存放在内存中的程序和数据彼此隔离、互不侵扰。操作系统中的虚存管理与处理器的MMU密切相关。

文件（File）

简单地说，文件就是存放在持久存储介质（比如硬盘、光盘、U盘等）上，方便应用程序和用户读写的数据。当处理器需要访问文件中的数据时，可通过操作系统把它们装入内存。放在硬盘上的程序也是一种文件。文件管理的任务是有效地支持文件的存储、检索和修改等操作。

操作系统特征

基于操作系统的四个抽象，我们可以看出，从总体上看，操作系统具有五个方面的特征：虚拟性（Virtualization）、并发性（concurrency）、异步性、共享性和持久性（persistency）。在虚拟性方面，可以从操作系统对内存，CPU的抽象和处理上有更好的理解；对于并发性和共享性方面，可以从操作系统支持多个应用程序“同时”运行的情况来理解；对于异步性，可以从操作系统调度，中断处理对应用程序执行造成的影响等几个放马来理解；对于持久性方面，可以从操作系统中的文件系统支持把数据方便地从磁盘等存储介质上存入和取出来理解。

虚拟性

内存虚拟化

首先来看看内存的虚拟化。程序员在写应用程序的时候，不用考虑其程序的起始内存地址要放到计算机内存的具体某个位置，而是用字符串符号定义了各种变量和函数，直接在代码中便捷地使用这些符号就行了。这是由于操作系统建立了一个地址固定，空间巨大的虚拟内存给应用程序来运行，这是空间虚拟化。这里的每个符号在运行时是要对应到具体的内存地址的。这些内存地址的具体数值是什么？程序员不用关心。为什么？因为编译器会自动帮我们吧这些符号翻译成地址，形成可执行程序。程序使用的内存是否占得太大了？在一般情况下，程序员也不用关心。

> 还记得虚拟地址（逻辑地址）的描述吗？
> 但编译器(compiler，比如gcc)和链接器（linker，比如ld）也不知道程序每个符号对应的地址应该放在未来程序运行时的哪个物理内存地址中。所以，编译器的一个简单处理办法就是，设定一个固定地址（比如 0x10000）作为起始地址，开始存放代码，代码之后是数据，所有变量和函数的符号都在这个起始地址之后的某个固定偏移位置。假定程序每次运行都是位于一个不会变化的起始地址。
> 这里的变量指的是全局变量，其地址在编译链接后会确定不变。但局部变量是放在堆栈中的，会随着堆栈大小的动态变化而变化。
> 这里编译器产生的地址就是虚拟地址。
> 这里，编译器和链接器图省事，找了一个适合它们的解决办法。当程序要运行的时候，这个符号到机器物理内存的映射必须要解决了，这自然就推到了操作系统身上。操作系统会把编译器和链接器生成的执行代码和数据放到物理内存中的空闲区域中，并建立虚拟地址到物理地址的映射关系。由于物理内存中的空闲区域是动态变化的，这也导致虚拟地址到物理地址的映射关系是动态变化的，需要操作系统来维护好可变的映射关系，确保编译器“固定起始地址”的假设成立。只有操作系统维护好了这个映射关系，才能让程序员只需写一些易于人理解的字符串符号来代表一个内存空间地址，且编译器只需确定一个固定地址作为程序的起始地址就可以生成一个不用考虑将来这个程序要在哪里运行的问题，从而实现了空间虚拟化。

应用程序在运行时不用考虑当前物理内存是否够用。如果应用程序需要一定空间的内存，但由于在某些情况下，物理内存的空闲空间可能不多了，这时操作系统通过把物理内存中最近没使用的空间（不是空闲的，只是最近用得少）换出（就是“挪地”）到硬盘上暂时缓存起来，这样空闲空间就大了，就可以满足应用程序的运行时内存需求了，从而实现了空间大小虚拟化。

CPU虚拟化

再来看CPU虚拟化。不同的应用程序可以在内存中并发运行，相同的应用程序也可有多个拷贝在内存中并发运行。而每个程序都“认为”自己完全独占了CPU在运行，这是”时间虚拟化“。这其实也是操作系统给了运行的应用程序一个虚拟幻象。其实是操作系统把时间分成小段，每个应用程序占用其中一小段时间片运行，用完这一时间片后，操作系统会切换到另外一个应用程序，让它运行。由于时间片很短，操作系统的切换开销也很小，人眼基本上是看不出的，反而感觉到多个程序各自在独立”并行“执行，从而实现了时间虚拟化。

> 并行（Parallel）是指两个或者多个事件在同一时刻发生；而并发（Concurrent）是指两个或多个事件在同一时间间隔内发生。
> 对于单CPU的计算机而言，各个”同时“运行的程序其实是串行分时复用一个CPU，任一个时刻点上只有一个程序在CPU上运行。
> 这些虚拟性的特征给应用程序的开发和执行提供了非常方便的环境，但也给操作系统的设计与实现提出了很多挑战。

并发性

操作系统为了能够让CPU充分地忙起来并充分利用各种资源，就需要给很多任务给它去完成。这些任务是分时完成的，有操作系统来完成各个应用在运行时的任务切换。并发性虽然能有效改善系统资源的利用率，但并发性也带来了对共享资源的争夺问题，即同步互斥问题；执行时间的不确定性问题，即并发程序在执行中是走走停停，断续推进的。并发性对操作系统的设计也带来了很多挑战，一不小心就会出现程序执行结果不确定，程序死锁等很难调试和重现的问题。

异步性

在这里，异步是指由于操作系统的调度和中断等，会不时地暂停或打断当前正在运行的程序，使得程序的整个运行过程走走停停。在应用程序运行的表现上，特别它的执行完成时间是不可预测的。但需要注意，只要应用程序的输入是一致的，那么它的输出结果应该是符合预期的。

共享性

共享是指多个应用并发运行时，宏观上体现出它们可同时访问同一个资源，即这个资源可被共享。但其实在微观上，操作系统在硬件等的支持下要确保应用程序互斥或交替访问这个共享的资源。比如两个应用同时写访问同一个内存单元，从宏观的应用层面上看，二者都能正确地读出同一个内存单元的内容。而在微观上，操作系统会调度应用程序的先后执行顺序，在数据总线上任何一个时刻，只有一个应用去访问存储单元。

持久性

操作系统提供了文件系统来从可持久保存的存储介质（硬盘，SSD等，以后以硬盘来代表）中取数据和代码到内存中，并可以把内存中的数据写回到硬盘上。硬盘在这里是外设，具有持久性，以文件系统的形式呈现给应用程序。

> 文件系统也可看成是操作系统对硬盘的虚拟化
> 这种持久性的特征进一步带来了共享属性，即在文件系统中的文件可以被多个运行的程序所访问，从而给应用程序之间实现数据共享提供了方便。即使掉电，磁盘上的数据还不会丢失，可以在下一次机器加电后提供给运行的程序使用。持久性对操作系统的执行效率提出了挑战，如何让数据在高速的内存和慢速的硬盘间高效流动是需要操作系统考虑的问题。

“麻雀“OS--uCore

为了学习OS，需要了解一个上百万代码的操作系统吗？自己写一个操作系统难吗？别被现在上百万行的Linux和Windows操作系统吓倒。当年Thompson乘他老婆带着小孩度假留他一人在家时，写了UNIX；当年Linus还是一个21岁大学生时完成了Linux雏形。站在这些巨人的肩膀上，我们能否也尝试一下做“巨人”的滋味呢？

MIT的Frans Kaashoek等在2006年参考PDP-11上的UNIX Version 6写了一个可在X86上跑的操作系统xv6（基于MIT License），用于学生学习操作系统。我们可以站在他们的肩膀上，基于xv6的设计，尝试着一步一步完成一个从“空空如也”到“五脏俱全”的“麻雀”操作系统—ucore，此“麻雀”包含虚存管理、进程管理、处理器调度、同步互斥、进程间通信、文件系统等主要内核功能，总的内核代码量（C+asm）不会超过5K行。充分体现了“小而全”的指导思想。

ucore的运行环境可以是真实的X86计算机，不过考虑到调试和开发的方便，我们可采用X86模拟器，比如QEMU、BOCHS等，或X86虚拟运行环境，比如VirtualBox、VMware Player等。ucore的开发环境主要是GCC中的gcc、gas、ld和MAKE等工具，也可采用集成了这些工具的IDE开发环境Eclipse-CDT。运行环境和开发环境既可以在Linux或Windows中使用。

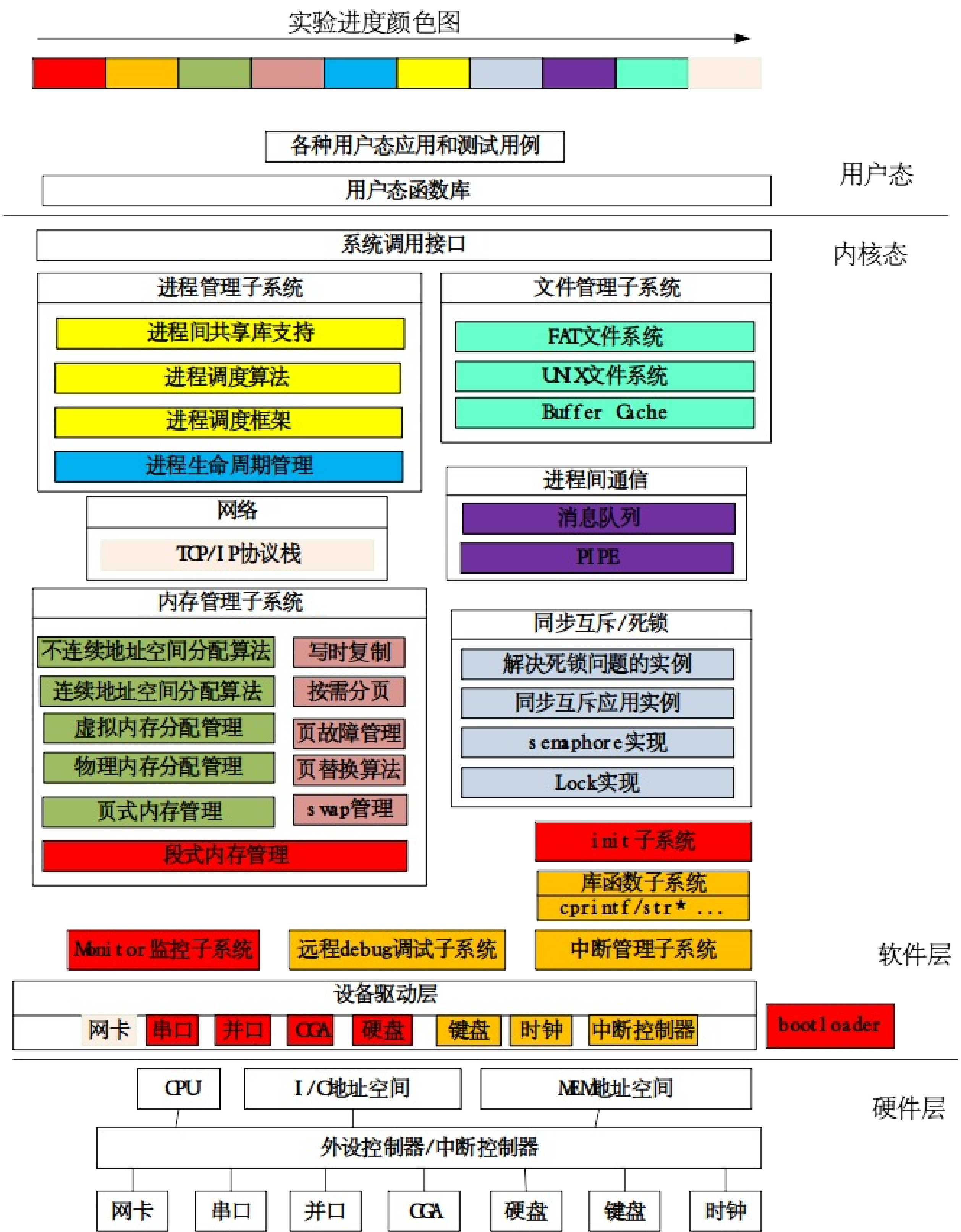
ucore简介

ucore目前支持的硬件环境是基于Intel 80386以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现ucore的过程逐渐展开介绍。那我们准备如何一步一步实现ucore呢？安装一个操作系统的开发过程，我们可以有如下的开发步骤：

1. bootloader+toy ucore：理解操作系统启动前的硬件状态和要做的准备工作，了解运行操作系统的外设硬件支持，操作系统如何加载到内存中，理解两类中断--“外设中断”，“陷阱中断”，内核态和用户态的区别；
2. 物理内存管理：理解x86分段/分页模式，了解操作系统如何管理物理内存；
3. 虚拟内存管理：理解OS虚存的基本原理和目标，以及如何结合页表+中断处理（缺页故障处理）来实现虚存的目标，如何实现基于页的内存替换算法和替换过程；
4. 内核线程管理：理解内核线程创建、执行、切换和结束的动态管理过程，以及内核线程的运行周期等；
5. 用户进程管理：理解用户进程创建、执行、切换和结束的动态管理过程，以及在用户态通过系统调用得到内核中各种服务的过程；
6. 处理器调度：理解操作系统的调度过程和调度算法；
7. 同步互斥与进程间通信：理解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，如何避免死锁，以及线程/进程间如何进行信息交换和共享；
8. 文件系统：理解文件系统的具体实现，与进程管理和内存管理等的关系，缓存对操作系统IO访问的性能改进，虚拟文件系统（VFS）、buffer cache和disk driver之间的关系。

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示

> （！可进一步标注处各个proj在下图中的位置）



实验进度颜色图
各种用户态应用和测试用例
用户态函数库
用户态
系统调用接口
内核态
进程管理子系统
进程间共享库支持
进程调度算法
进程调度框架
进程生命周期管理
文件管理子系统
FAT文件系统
UNIX文件系统
Buffer Cache
网络
TCP/IP协议栈
进程间通信
消息队列
PIPE
内存管理子系统
不连续地址空间分配算法
写时复制
连续地址空间分配算法
按需分页
虚拟内存分配管理
页故障管理
物理内存分配管理
页替换算法
页式内存管理
swap管理
段式内存管理
同步互斥/死锁
解决死锁问题的实例
同步互斥应用实例
semaphore实现
Lock实现
init子系统
库函数子系统
cprintf/str* ...
Monitor监控子系统
远程debug调试子系统
中断管理子系统
软件层
设备驱动层
网卡
串口
并口
CGA
硬盘
键盘
时钟
中断控制器
bootloader
CPU
I/O地址空间
MEM地址空间
硬件层
外设控制器/中断控制器
网卡
串口
并口
CGA
硬盘
键盘
时钟

本章比较概要地介绍了操作系统运行的计算机硬件架构，包括CPU、内存和外设，并对操作系统的历史发展、定义、目标、接口、抽象和特征等进行了阐述。最后简要介绍了课程实验用到的ucore操作系统。

启动操作系统

用一句话描述本章

站在操作系统的最底层，了解操作系统的启动，与物理硬件：CPU，内存和多种外设实现“零距离”接触，看到它们并管理它们！

本章收获的知识

- 与操作系统原理相关
    - I/O设备管理：涉及程序循环检测方式和中断启动方式、I/O地址空间
    - 内存管理：基于分段机制的内存管理
    - 异常处理：涉及中断、故障和陷阱
    - 特权级：内核态和用户态
- 计算机系统和编程
    - 硬件
        - PC从加电到加载操作系统内核的整个过程
        - OS内核在内存中的布局
        - 并口访问、串口访问、CGA字符显示、硬盘数据访问、时钟访问
    - 软件
        - ELF执行文件格式
        - 栈的实现并实现函数调用栈跟踪函数
        - 调试操作系统

本章涉及的实验

读者通过阅读本章的内容并动手实践相关的6个实验项目：

- proj1：能够切换到保护模式并显示字符的bootloader
- proj2/3：可读ELF格式文件的bootloader和显示字符的ucore
- proj3.1：内置监控自身运行状态的ucore
- proj4：可管理中断和处理基于中断的键盘/时钟的ucore
- proj4.1.1：支持通过中断方式的内核态/用户态切换的ucore

本章概述

其实这一章的内容与操作系统原理相关的部分较少，与计算机体系结构（特别是x86）的细节相关的部分较多。但这些内容对写一个操作系统关系较大，要知道操作系统是直接与硬件打交道的软件，所以它需要"知道"需要硬件细节，才能更好地控制硬件。另一方面，部分内容涉及到操作系统的重要抽象--中断类异常，能够充分理解中断类异常为以后进一步了解进程切换、上下文切换等概念会很有帮助。

本章的实验内容涉及的是写一个bootloader能够启动一个操作系统--ucore。在完成bootloader的过程中，逐渐增加bootloader和ucore的能力，涉及x86处理器的保护模式切换、解析ELF执行文件格式等，这对于理解操作系统的加载过程以及在操作系统在内存中的位置、内存管理、用户态与内核态的区别等有帮助。而相关project中bootloader和操作系统本身的字符显示的I/O处理、读硬盘数据的I/O处理、键盘/时钟的中断处理等内容，则是操作系统原理中一般在靠后位置提到的设备管理的实际体现。纵观操作系统的发展史，从早期到现在的操作系统主要功能之一就是完成繁琐的I/O处理，给上层应用提供比较简洁的I/O服务，屏蔽硬件处理的复杂性。这也是操作系统的虚拟机功能的体现。另外，本章还介绍了对硬件模拟器的使用，对操作系统的panic处理和远程debug功能的支持，这样有助于读者能够方便地分析操作系统中的错误和调试操作系统。由于本章涉及的硬件知识较多，无疑增大了读者的阅读难度，需要读者在结合阅读本章并实际动手实验来进行深入理解。

显示字符的toy bootloader

实验目标

操作系统是一个软件，也需要通过某种手段加载并运行它。在这里我们将通过另外一个更加简单的软件-bootloader来完成这些工作。为此，我们需要完成一个能够切换到x86的保护模式并显示字符的bootloader，为将来启动操作系统做准备。proj1提供了一个非常小的bootloader，整个bootloader的大小小于512个字节，这样才能放到硬盘的主引导扇区中。

> 这里对x86的保护模式不必太在意，后续会进一步讲解

通过分析和实现这个bootloader，读者可以了解到：

- 与操作系统原理相关
  - I/O设备管理：设备管理的基本概念，涉及简单的信息输出
  - 内存管理：基于分段机制的存储管理，x86的实模式/保护模式以及切换到保护模式的方法
- 计算机系统和编程
  - 硬件
    - PC加电后启动bootloader的过程
    - 通过串口/并口/CGA输出字符的方法
  - 软件
    - bootloader的文件组成
    - 编译运行bootloader的过程
    - 调试bootloader的方法
    - 在汇编级了解栈的结构和处理过程

proj1概述

实现描述

proj1实现了一个简单的bootloader，主要完成的功能是初始化寄存器内容，完成实模式到保护模式的转换，在保护模式下通过PIO方式控制串口、并口和CGA等进行字符串输出。

项目组成

[要点（非OSP）：bootloader的编译生成过程] lab1中包含的第一个工程小例子是proj1：一个可以切换到保护模式并显示字符串的bootloader。proj1的整体目录结构如下所示：

```
proj1 /
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- libs
|   |-- types.h
|   `-- x86.h
|-- Makefile
`-- tools
    |-- function.mk
    |-- gdbinit
    `-- sign.c

3 directories, 9 files
```

其中一些比较重要的文件说明如下：

- bootasm.S ：定义并实现了bootloader最先执行的函数start，此函数进行了一定的初始化，完成了从实模式到保护模式的转换，并调用bootmain.c中的bootmain函数。
- bootmain.c：定义并实现了bootmain函数实现了通过屏幕、串口和并口显示字符串。
- asm.h：是bootasm.S汇编文件所需要的头文件，主要是一些与X86保护模式的段访问方式相关的宏定义。
- types.h：包含一些无符号整型的缩写定义。
- x86.h：一些用GNU C嵌入式汇编实现的C函数（由于使用了inline关键字，所以可以理解为宏）。
- Makefile和function.mk：指导make完成整个软件项目的编译，清除等工作。
- sign.c：一个C语言小程序，是辅助工具，用于生成一个符合规范的硬盘主引导扇区。
- gdbinit：用于gdb远程调试的初始命令脚本

从中，我们可以看出bootloader主要由bootasm.S和bootmain.c组成，当你完成编译后，你会发现这个bootloader只有区区的3百多字节。下面是编译运行bootloader的过程。

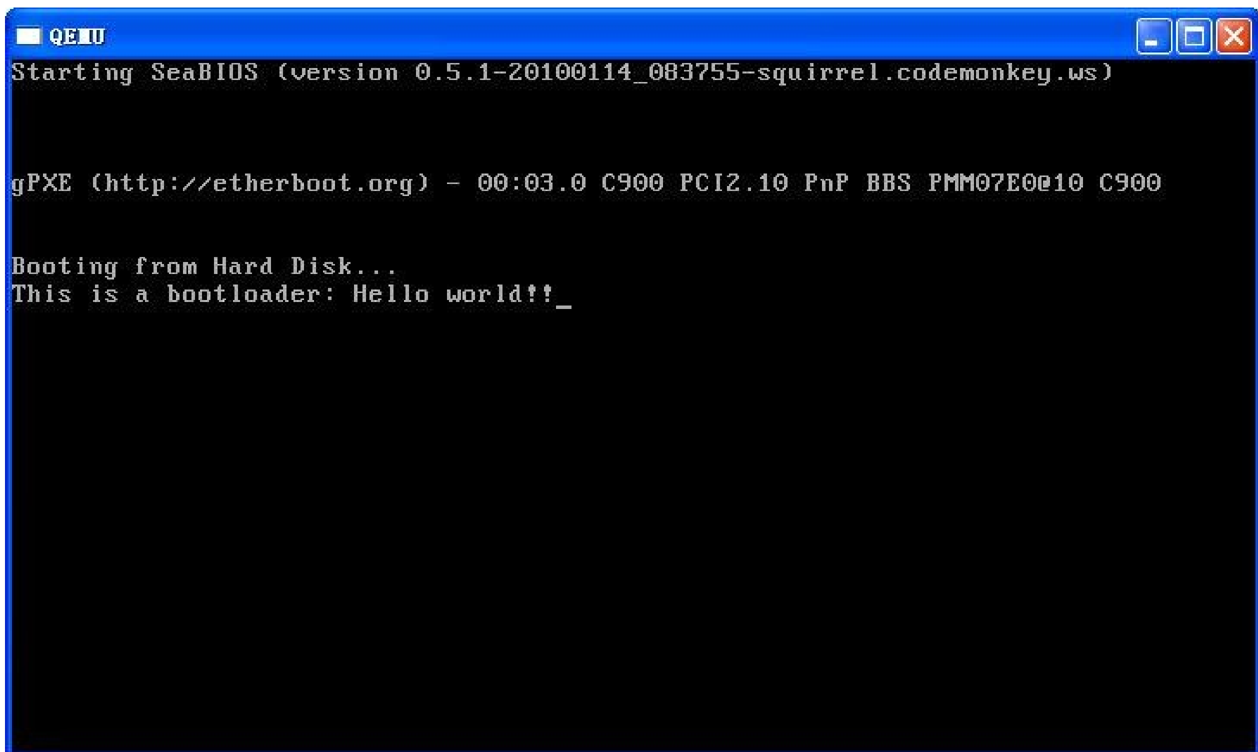
> 【提示】bootloader是一个超小的系统软件，在功能上与我们一般的应用软件不同，主要用于硬件简单初始化和加载运行操作系统。在编写bootloader的时候，需要了解它所处的硬件环境（比如它在内存中的起始地址，它的储存空间的位置和大小限制等）。而这些是编写应用软件不太需要了解的，因为操作系统和编译器帮助应用软件考虑了这些问题。

编译运行

【实验 编译运行**bootloader**】

在proj1下执行make，在proj1/bin目录下可生成一个ucore.img。ucore.img是一个包含了bootloader或OS的硬盘镜像，通过执行如下命令可在硬件虚拟环境 qemu中运行bootloader或OS：

```
make                    //生成bootloader和对应的主引导扇区
make qemu          //通过qemu硬件模拟器来运行bootloader
make clean          //清除生成的临时文件,bootloader和对应的主引导扇区
```



我们除了需要了解bootloader的功能外，还需要进一步了解bootloader的编译链接和最终执行码的生成过程，从而能够清楚生成的代码是否是我们所期望的。proj1中的Makefile是一个配置脚本，make软件工具能够通过Makefile完成管理bootloader的C/ASM代码生成执行码的整个过程。Makefile的内容比较复杂，不过读者在前期只需会执行make [参数]来生成代码和清除代码即可。对于本实验的make的执行过程如下所示：

```
    1. gcc -O2 -o tools/sign tools/sign.c
    2. i386-elf-gcc -fno-builtin -Wall -MD -ggdb -m32 -fno-stack-protector -O -nostdin
c -Iinclude -Iinclude/x86 -c bootloader/bootmain.c -o obj/bootmain.o
    3. i386-elf-gcc -fno-builtin -Wall -MD -ggdb -m32 -fno-stack-protector -nostdinc -
Iinclude -Iinclude/x86 -c bootloader/bootasm.S -o obj/bootasm.o
    4. i386-elf-ld  -N -e start -Ttext 0x7C00 -o obj/bootblock.o obj/bootasm.o obj/boo
tmain.o
    5. i386-elf-objdump -S obj/bootblock.o > obj/bootblock.asm
    6. i386-elf-objcopy -S -O binary obj/bootblock.o obj/bootblock.out
    7. sign.exe obj/bootblock.out obj/bootblock
       obj/bootblock.out size: 380 bytes
       build 512 bytes boot sector: obj/bootblock success!
    8. dd if=/dev/zero of=obj/ucore.img count=10000
       10000+0 records in
       10000+0 records out
       5120000 bytes (5.1 MB) copied, 0.509 s, 10.1 MB/s
    9. dd if=obj/bootblock of=obj/ucore.img conv=notrunc
       1+0 records in
       1+0 records out
       512 bytes (512 B) copied, 0.011 s, 46.5 kB/s
```

这**9**步的含义是：

```
1. 编译生成sign执行程序，用于生成一个符合规范的硬盘主引导扇区；
2. 用gcc编译器编译bootmain.c，生成ELF格式的目标文件bootmain.o；
3. 用gas汇编器（gcc只是一个包装）编译bootasm.S，生成ELF格式的目标文件bootasm.o；
4. 用ld链接器把bootmain.o和bootasm.o链接在一起，形成生成ELF格式的执行文件bootblock.o；
5. 目标文件信息导出工具objdump反汇编bootblock.o，生成bootblock.asm，通过查看bootlock.asm内容，
可以了解bootloader的实际执行代码；
6. 文件格式转换和拷贝工具objcopy把ELF格式的执行文件bootblock.o转换成binary格式的执行文件bootbloc
k.out；
7. 通过sign执行程序，把bootblock.out（本身大小需要小于510字节）扩展到512字节，形成一个符合规范的硬
盘主引导扇区bootblock;
8. 设备级转换与拷贝工具dd生成一个内容都为“0”的磁盘文件ucore.img；
9. 设备级转换与拷贝工具dd进一步把bootblock覆盖到ucore.img的前512个字节空间中，这样就可以把ucore.i
mg作为一个可启动的硬盘被硬件模拟器qemu使用。
```

如果需要了解Makefile中的内容，需要进一步看看附录“ucore实验中的常用工具”一节。

【背景】Intel 80386加电后启动过程

【要点（非**OSP**）：**80836**物理内存地址空间】

【要点（非**OSP**）：**80836**加电后的第一条指令位】

大家一般都知道bootloader负责启动操作系统，但bootloader自身是被谁加载并启动的呢？为了追根溯源，我们需要了解当计算机加电启动后，到底发生了什么事情。

对于绝大多数计算机系统而言，操作系统和应用软件是存放在磁盘（硬盘/软盘）、光盘、EPROM、ROM、Flash等可在掉电后继续保存数据的存储介质上。当计算机加电后，一般不直接执行操作系统，而是一开始会到一个特定的地址开始执行指令，这个特定的地址存放了系统初始化软件，通过执行系统初始化软件（可固化在ROM或Flash中，也称firmware，固件）完成基本I/O初始化和引导加载操作系统的功能。简单地说，系统初始化软件就是在操作系统内核运行之前运行的一段小软件。通过这段小软件的基本I/O初始化部分，我们可以初始化硬件设备、建立系统的内存空间映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。最终系统初始化软件的引导加载部分把操作系统内核映像加载到RAM中，并将系统控制权传递给它。

对于基于Intel 80386的计算机而言，其中的系统初始化软件由BIOS (Basic Input Output System，即基本输入/输出系统，其本质是一个固化在主板Flash/CMOS上的软件)和位于软盘/硬盘引导扇区中的OS Boot Loader（在ucore中的bootasm.S和bootmain.c）一起组成。BIOS实际上是被固化在计算机ROM（只读存储器）芯片上的一个特殊的软件，为上层软件提供最底层的、最直接的硬件控制与支持。

以基于Intel 80386的计算机为例，计算机加电后，整个物理地址空间如下图所示：

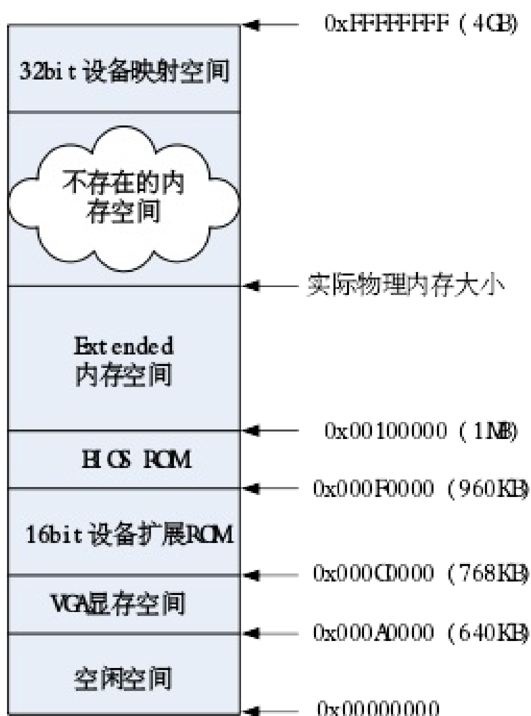
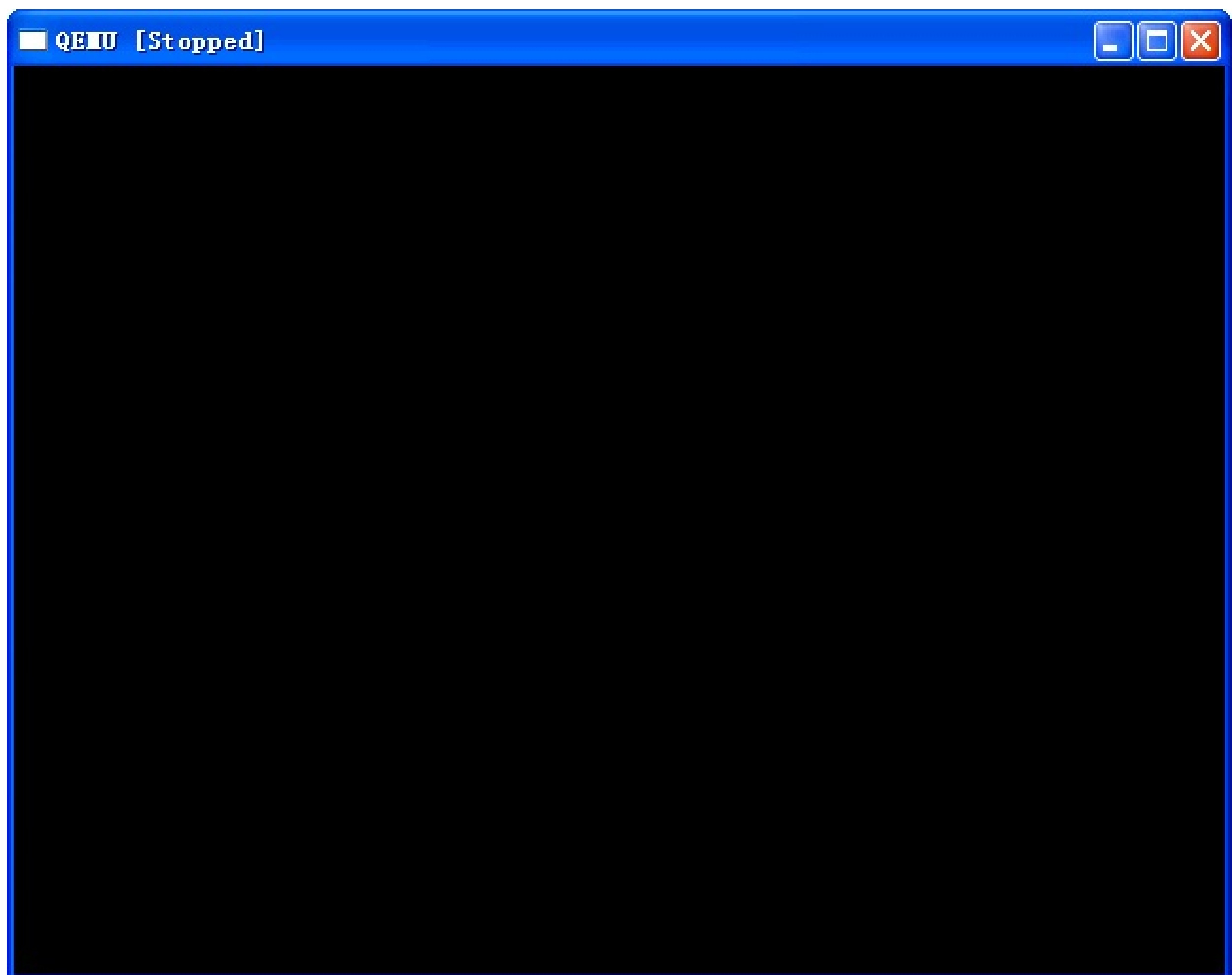


图2-1 基于Intel 80386的计算机物理地址空间

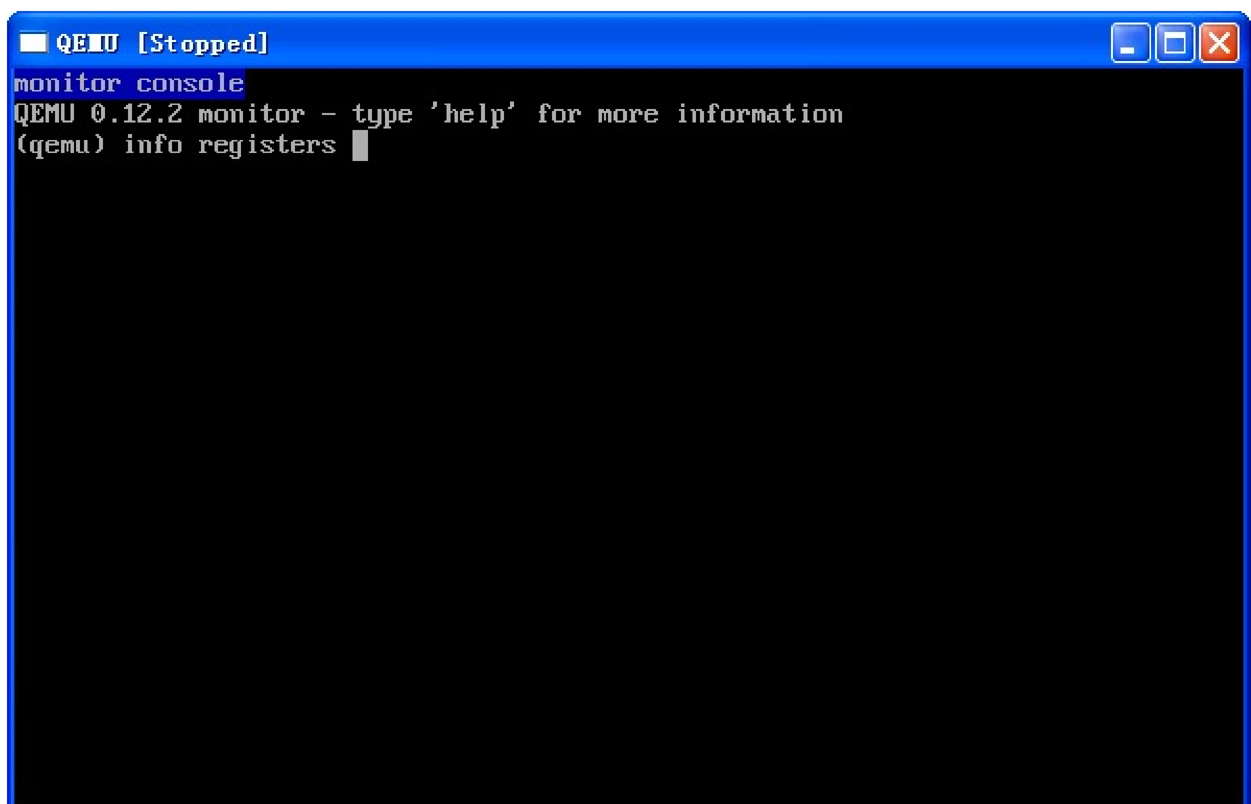
处理器处于实模式状态（在86386中，段机制一直存在，可进一步参考2.1.5 【背景】理解保护模式和分段机制），从物理地址0xFFFFFFF0开始执行。初始化状态的CS和EIP确定了处理器的初始执行地址，此时CS中可见部分-选择子（selector）的值为0xF000，而其不可见部分-基地址（base）的值为0xFFFF0000；EIP的值是0xFFF0，这样实际的线性地址（由于没有启动也机制，所以线性地址就是物理地址）为CS.base+EIP=0xFFFFFFF0。在0xFFFFFFF0这里只是存放了一条跳转指令，通过跳转指令跳到BIOS例行程序起始点。更详细的解释可以参考文献[1]的第九章的9.1节“INITIALIZATION OVERVIEW”。另外，我们可以通过硬件模拟器qemu来进一步认识上述结果。

实验**2-1**：通过**qemu**了解**Intel 80386**启动后的**CS**和**EIP**值，并分析第一条指令的内容

1. 启动qemu并让其停到执行第一条指令前，这需要增加一个参数”-S” qemu –S
2. 这是qemu会弹出一个没有任何显示内容的图形窗口，显示如下：



1. 然后通过按"Ctrl+Alt+2"进入qemu的monitor界面，为了了解80386此时的寄存器内容，在monitor界面下输入命令 "info registers"



1. 可获得intel 80386启动后执行第一条指令前的寄存器内容，如下图所示

```
QEMU [Stopped]
EIP=0000fff0 EFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300
CS =f000 ffff0000 0000ffff 00009b00
SS =0000 00000000 0000ffff 00009300
DS =0000 00000000 0000ffff 00009300
FS =0000 00000000 0000ffff 00009300
GS =0000 00000000 0000ffff 00009300
LDT=0000 00000000 0000ffff 00008200
TR =0000 00000000 0000ffff 00008b00
GDT=     00000000 0000ffff
IDT=     00000000 0000ffff
CR0=60000010 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000 XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000 XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=00000000000000000000000000000000
(qemu)
```

从上图中，我们可以看到EIP=0xfff0，CS的selector=0xf000，CS的base=0xfff0000。

BIOS做完计算机硬件自检和初始化后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区(即主引导扇区或启动扇区)到内存一个特定的地址0x7c00处，然后CPU控制权会转移到那个地址继续执行。至此BIOS的初始化工作做完了，进一步的工作交给了ucore的bootloader；ucore的bootloader会完成处理器从实模式到保护模式的转换，并从硬盘上读取并加载ucore。其大致流程如下图所示：

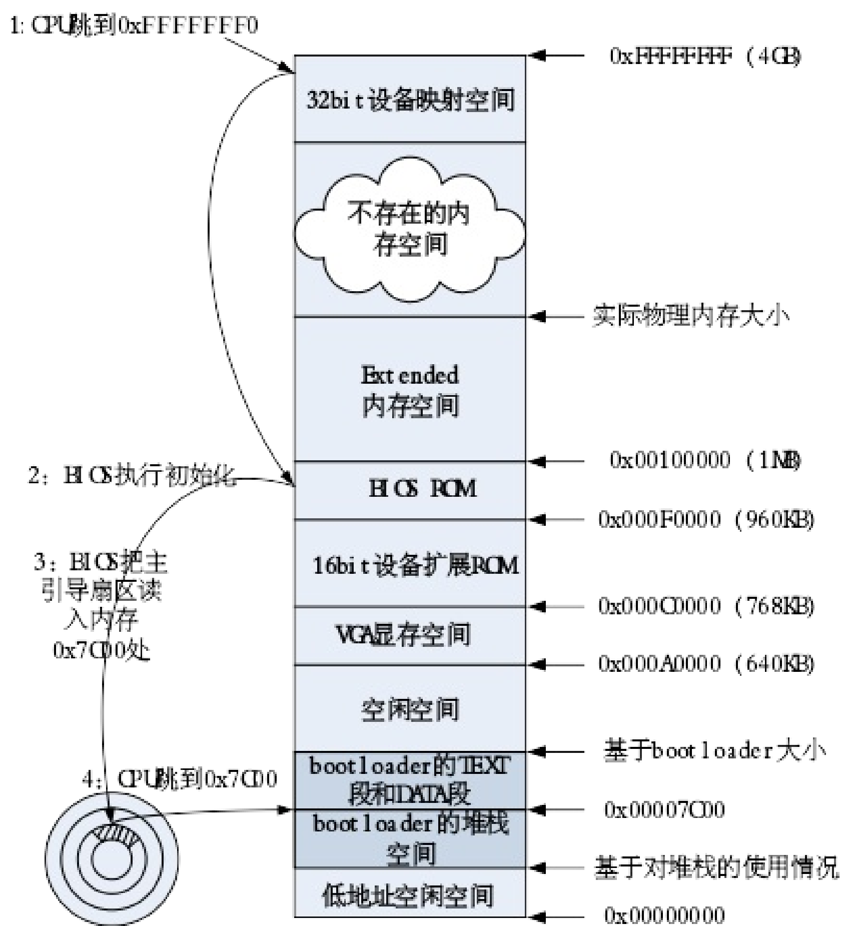


图2-2 Intel80386启动过程

【背景】设备管理：理解设备访问机制

在本章涉及的bootloader和ucore都需要对I/O设备进行访问，比如通过串口、并口和CGA显示器显示字符串，读取硬盘数据，处理时钟中断等，已经需要读者用到操作系统的I/O设备管理知识了。为此，我们需要操作系统的设备管理进行一个简要描述。

在计算机系统中，操作系统需要管理各种设备，即给它们发送控制命令、捕获中断、错误处理等；为此专门设置了一个子系统：设备管理子系统来完成这些琐碎的工作。同时设备管理子系统还需要提供一个简单易用的统一接口，并尽可能地使其他内核功能组件或应用可通过这个统一接口访问所有的设备，即实现与设备的无关性。比如在proj1中，bootloader提供了一个显示字符的函数接口cons_putc（位于bootmain.c中），在proj3中的提供了一个显示格式化信息的函数接口cprintf（位于printf.c中），这样操作系统的其他功能组件就可以直接使用这些简单易用的接口来输出信息，而不是通过繁琐的I/O命令与具体的设备打交道。cprintf的实现相对复杂，用到C语言的可变列表参数等，大家只要把它的功能理解为C语言应用库中的printf的简化版即可，并掌握最后是如何通过调用cons_putc函数完成具体的I/O字符输出。

接下来，我们将从操作系统概念的角度对I/O设备组成、控制设备的方式进行阐述，并进一步对实验中所使用的基于Programmed I/O （PIO）方式访问并口、CGA和硬盘进行具体分析。

硬件设备简介

对于硬件设备而言，操作系统所关心的并不是硬件自身的设计，而是如何来对它进行控制，即该硬件所接受的控制命令、所完成的功能，以及所返回的出错。所以在设计操作系统的设备管理子系统时，需要了解计算机系统中I/O总线上连接的I/O控制器（比如PC机中的CGA控制器、串口控制器、并口控制器、时钟控制器8254，中断控制器8259等）。

I/O控制器在物理上包含三个层次：I/O地址空间、I/O接口和设备控制器。每个连接到I/O总线上的设备都有自己的I/O地址空间（即I/O端口），这也是CPU可以直接访问的地址。在 PC机中，支持基于I/O的I/O地址空间(通过IN/OUT这类的I/O访问指令访问)，也支持基于内存的I/O地址空间（通过MOV等访存指令访问）。这些I/O访问请求通过I/O总线传递给I/O接口。

I/O接口是处于一组I/O端口和对应的设备控制器之间的一种硬件电路。它将I/O访问请求中的特定值转换成设备所需要的命令和数据；并且检测设备的状态变化，及时将各种状态信息写回到特定I/O地址空间，供操作系统通过I/O访问指令来访问。I/O接口包括键盘接口、图形接口、磁盘接口、总线鼠标、网络接口、括并口、串口、通用串行总线、PCMCIA接口和SCSI接口等。

设备控制器并不是所有I/O设备所必须的，只有少数复杂的设备才需要。它负责解释从I/O接口接收到的高级命令，并将其以适当的方式发送到I/O设备；并且对I/O设备发送的消息进行解释并修改I/O端口的状态寄存器。典型的设备控制器就是磁盘控制器，它将CPU发送过来的读写

数据指令转换成底层的磁盘操作。

控制设备的方式

操作系统对硬件设备的控制方式主要与三种：程序循环检测方式(Programmed I/O，简称PIO)、中断驱动方式(Interrupt-driven I/O)、直接内存访问方式(DMA, Direct Memory Access)。

在本章的proj1实验中，bootloader需要显示字符串，就是采用相对简单的PIO方式。PIO方式是一种通过CPU执行I/O端口指令来进行数据读写的数据交换模式，被广泛应用于硬盘、光驱等设备的基础传输模式中。这种I/O访问方式使用CPU I/O端口指令来传送所有的命令、状态和数据，需要处理器全程参与，效率较低，但编程很简单。后面讲到的中断方式和直接内存访问（Direct Memory Access，DMA）方式将更加高效。

对于程序循环检测方式而言，其控制方式体现在执行过程中通过不断地检测I/O设备的当前状态，来控制I/O操作。具体而言，在进行I/O操作之前，要循环地检测设备是否就绪；在I/O操作进行之中，要循环地检测设备是否已完成；在I/O操作完成之后，还要把输入的数据保存到内存（输入操作）。从硬件的角度来说，控制I/O的所有工作均由CPU来完成。所以此方式也称为繁忙等待方式（busy waiting）或轮询方式（polling）。其缺点是在进行I/O操作时，一直占用CPU时间。

中断驱动方式的基本思路是用户任务通过系统调用函数来发起I/O操作。执行系统调用后会阻塞该任务，调度其他的任务使用CPU。在I/O操作完成时，设备向CPU发出中断，然后在中断服务例程中做进一步的处理。在中断驱动方式下，数据的每次读写还是通过CPU来完成。但是当I/O设备在进行数据处理时，CPU不必等待，可以继续执行其他的任务。采用这种方式可提供CPU利用率。编程方面，要考虑异步特性，相对麻烦一些。

使用DMA的控制方式，首先需要有DMA控制器。该控制器可集成在设备控制器中，也可集成在主板上。DMA控制器可以直接去访问系统总线，它能代替CPU指挥I/O设备与内存之间的数据传送，在执行完毕后再通知CPU。这种方式可大大减少CPU的执行开销，适合大数据量的设备数据传送。在编程方面，需要对DMA进行编程和异步中断编程，相对更加复杂一些。

串口（serial port）访问控制

串口是一个字符设备，proj1通过串口输出需要显示的信息。考虑到简单性，在proj1中没有对串口设备进行初始化，通过串口进行输出的过程也很简单：第一步：执行inb指令读取串口的I/O地址（COM1 + COM_LSR）的值，如果发现发现读出的值代表串口忙，则空转一小会（0x84是什么地址???）；如果发现发现读出的值代表串口空闲，则执行outb指令把字符写到串口的I/O地址（COM1 + COM_TX），这样就完成了一个字符的串口输出。在proj1的bootmain.c中的serial_putc函数完成了串口输出字符的工作,可参看其函数来了解大致实现。有关串口的硬件细节可参考附录 补充材料。

并口（parallel port）访问控制

并口也是一个字符设备，proj1也通过并口输出需要显示的信息。考虑到简单性，在proj1中没有对并口设备进行初始化，通过并口进行输出的过程也很简单：第一步：执行inb指令读取并口的I/O地址（LPTPORT + 1）的值，如果发现发现读出的值代表并口忙，则空转一小会再读；如果发现发现读出的值代表并口空闲，则执行outb指令把字符写到并口的I/O地址（LPTPORT ），这样就完成了一个字符的并口输出。在proj1的bootmain.c中的lpt_putc函数完成了并口输出字符的工作,可参看其函数来了解大致实现。有关并口的硬件细节可参考附录补充材料。

CGA字符显示控制

彩色图形适配器（Color Graphics Adapter，CGA）支持7种彩色和文本/图形显示方式，proj1也通过CGA进行信息显示。在80列×25行的文本字符显示方式下，有单色和16色两种显示方式。CGA显示控制器标配有16KB显示内存（占用内存地址范围0xb8000～0xbc000），可以看成是一种内存块设备，即bootloader和操作系统可以直接对显存进行内存访问，从而完成信息显示。在CGA显示控制器中，字符显示内存从线性地址0x000B8000开始，在80列×25行的范围内，共2000字符。每个字符需要两个字节来显示：第一个字节是想要显示的字符 ，第二个字节用来确定前景色和背景色。前景色用低4位（0~3位）来表示，背景色用第4位到第6位来表示。最高位表示这个字符是否闪烁，1表示闪烁，0表示不闪烁。

如果要在屏幕上设置光标，则它须通过CGA显示控制器的I/O端口开控制。显示控制索引寄存器的I/O端口地址为0x3d4；数据寄存器I/O端口地址为0x3d5。CGA显示控制器内部有一系列寄存器可以用来访问其状态。0x3d4和0x3d5两个端口可以用来读写CGA显示控制器的内部寄存器。方法是先向0x3d4端口写入要访问的寄存器编号，再通过0x3d5端口来读写寄存器数据。存放光标位置的寄存器编号为14和15。两个寄存器合起来组成一个16位整数，这个整数就是光标的位置。比如0表示光标在第0行第0列，81表示第1 行第1列（设屏幕共有80列）。

在proj1中没有对CGA显示控制器进行初始化，通过CGA显示控制器进行输出的过程也很简单：首先通过in/out指令获取当前光标位置；然后根据得到的位置计算出显存的地址，直接通过访存指令写内存来完成字符的输出；最后通过in/out指令更新当前光标位置。在proj1的bootmain.c中的cga_putc函数完成了CGA字符方式在某位置输出字符的工作，可参看其函数了解大致实现。

设备管理封装

proj1把上述三种设备进行了一个封装，提供了一个cons_puts函数接口：完成字符串的输出；和一个cons_putc函数接口，完成字符的输出。其他内核功能模块只需调用cons_puts或cons_putc就可完成向上述三个设备进行字符输出的功能。这也就体现了设备管理子系统提供

一个简单易用的统一接口的操作系统设计思想。

【背景】内存管理：理解保护模式和分段机制

为何要了解Intel 80386的保护模式和分段机制？首先，我们知道Intel 80386只有在进入保护模式后，才能充分发挥其强大的功能，提供更好的保护机制和更大的寻址空间，否则仅仅是一个快速的8086而已。没有一定的保护机制，任何一个应用软件都可以任意访问所有的计算机资源，这样也就无从谈起操作系统设计了。且Intel 80386的分段机制一直存在，无法屏蔽或避免。其次，在我们的bootloader设计中，涉及到了从实模式到保护模式的处理，我们的操作系统功能（比如分页机制）是建立在Intel 80386的保护模式上来设计的。如果我们不了解保护模式和分段机制，则我们面向Intel 80386体系结构的操作系统设计实际上是建立在一个空中楼阁之上。

实模式

80386的实模式是为了与8086处理器兼容而设置的。在实模式下，80386处理器就相当于一个快速的8086处理器。80386处理器被复位或加电的时候以实模式启动。这时候处理器中的各寄存器以实模式的初始化值工作。80386处理器在实模式下的存储器寻址方式和8086基本一致，由段寄存器的内容乘以16作为基地址，加上段内的偏移地址形成最终的物理地址，这时候它的32位地址线只使用了低20位，即可访问1MB的物理地址空间。在实模式下，80386处理器不能对内存进行分页机制的管理，所以指令寻址的地址就是内存中实际的物理地址。在实模式下，所有的段都是可以读、写和执行的。实模式下80386不支持优先级，所有的指令相当于工作在特权级（即优先级0），所以它可以执行所有特权指令，包括读写控制寄存器CR0等。这实际上使得在实模式下不太可能设计一个有保护能力的操作系统。实模式下不支持硬件上的多任务切换。实模式下的中断处理方式和8086处理器相同，也用中断向量表来定位中断服务程序地址。中断向量表的结构也和8086处理器一样，每4个字节组成一个中断向量，其中包括两个字节的段地址和两个字节的偏移地址。应用程序可以任意修改中断向量表的内容，使得计算机系统容易受到病毒、木马等的攻击，整个计算机系统的安全性无法得到保证。

【历史：寻址空间：**A20**地址线与处理器向下兼容】

Intel早期的8086 CPU提供了20根地址线,可寻址空间范围即0~2^20(00000H~FFFFFH)的1MB内存空间。但8086的数据处理位为16位,无法直接寻址1MB内存空间，所以8086提供了段地址加偏移地址的地址转换机制，就是我们常见的”段地址(16位):偏移地址(16位或有效地址)”,实际的计算方法为:”段地址*0x10H+偏移地址”，作为段地址的数据是放在段寄存器中的(16位)，而作为位偏移地址的数据则是通过8086提供的寻址方式来计算而来的(16位)。而“段值：偏移”这种表示法能够表示的最大内存为0x10FFEEH(即0xFFFF0H + 0xFFFFH)，所以当寻址到超过1MB的内存时，会发生“回卷”（不会发生异常）。但下一代的基于Intel 80286 CPU的PC AT计算机系统提供了24根地址线，这样CPU的寻址范围变为 2^24=16M,同时也提

供了保护模式，可以访问到1MB以上的内存了，此时如果遇到“寻址超过1MB”的情况，系统不会再“回卷”了，这就造成了向下不兼容。为了保持完全的向下兼容性，IBM决定在PC AT计算机系统上加个硬件逻辑，来模仿以上的回绕特征。他们的方法就是把A20地址线控制和键盘控制器的一个输出进行AND操作，这样来控制A20地址线的打开（使能）和关闭（屏蔽\禁止）。一开始时A20地址线控制是被屏蔽的（总为0），直到系统软件通过一定的I/O操作去打开它（参看bootloader的bootasm.S文件）。 当A20 地址线控制禁止时，则程序就像在8086中运行，1MB以上的地是不可访问的。在保护模式下A20地址线控制是要打开的。为了使能所有地址位的寻址能力,必须向键盘控制器8042发送一个命令。键盘控制器8042将会将它的的某个输出引脚的输出置高电平,作为 A20 地址线控制的输入。一旦设置成功之后,内存将不会再被绕回(memory wrapping),这样我们就可以寻址intel 80286 CPU支持的16M 内存空间，或者是寻址intel 80386 以上级别CPU支持的所有 4G内存空间了。 8042键盘控制器的I/O端口是0x60～0x6f，实际上IBM PC/AT使用的只有0x60和0x64两个端口（0x61、0x62和0x63用于与XT兼容目的）。8042通过这些端口给键盘控制器或键盘发送命令或读取状态。输出端口P2用于特定目的。位0（P20引脚）用于实现CPU复位操作，位1（P21引脚）用户控制A20信号线的开启与否。系统向输入缓冲（端口0x64）写入一个字节，即发送一个键盘控制器命令。可以带一个参数。参数是通过0x60端口发送的。 命令的返回值也从端口 0x60去读。 在proj1的bootasm.S中，“seta20.1”标号和“seta20.2”标号后的汇编代码即是用来完成A20地址线控制打开工作的。

保护模式概述

简单地说，通过保护模式，可以把虚拟地址空间映射到不同的物理地址空间，且在超出预设的空间范围会报错（一种保护机制的体现），且可以保证处于低特权级的代码无法访问搞特权级的数据（另外一种保护机制的体现）。 只有在保护模式下，80386的全部32位地址才能有效，可寻址高达4G字节的线性地址空间和物理地址空间，可访问64TB（有2^14个段，每个段最大空间为2^32字节）的虚拟地址空间，可采用分段存储管理机制和分页存储管理机制。这不仅为存储共享和保护提供了硬件支持，而且为实现虚拟存储提供了硬件支持。通过提供4个特权级和完善的特权检查机制，既能实现资源共享又能保证代码数据的安全及任务隔离。在保护模式下，特权级总共有4个，编号从0（最高特权）到3（最低特权）。有3种主要的资源受到保护：内存，I/O地址空间以及执行特殊机器指令的能力。在任一时刻，intel 80386 CPU都是在一个特定的特权级下运行的，从而决定了代码可以做什么，不可以做什么。这些特权级经常被称为为保护环（protection ring），最内的环（ring 0）对应于最高特权0，最外面的环（ring 3）一般给应用程序使用，对应最低特权3。在ucore中，CPU只用到其中的2个特权级：0（内核态）和3（用户态）。在保护模式下，我们可以通过查看CS寄存器的最低两位来了解当前正在运行的处理器是处于哪个特权级。

分段机制的地址转换

intel 80386 CPU提供了分段机制和分页机制两种内存管理方式，在当前计算机系统中是否需要这两种机制共存没有一个明确的答案，二者有它们各自独特的功能。在intel 80386 CPU中，只要进入保护模式，必然需要启动分段机制，且一直存在下去（分页不一定要一直存在），所以我们需要了解分段机制的原理。分段机制体现了内存中不同地址的一种转换/映射方式，即程序员编程所使用的地址（逻辑地址）和实际计算机中的物理地址需要通过分段机制来建立映射关系。分段机制将内存划分成以起始地址和长度限制这两个参数表示的内存块，这些内存块就称之为段（Segment）。编译器把源程序编译成执行程序时用到的代码段、数据段、堆和栈等概念在这里可以与段联系起来，二者在含义上是一致的。从操作系统原理上看，编译器实际上采用了基于分段的虚存管理方式来生成执行程序的，即应用程序员看到的逻辑地址和位于计算机上的物理地址之间有映射关系，二者可以是不同的。当然，后续章节中，我们还将介绍分页机制，即另一种使用更加广泛的地址转换/映射方式，这是操作系统实现虚存管理的重要基础。 简单地说，当CPU执行一条访存指令时（一个具体的指令），基于分段模式的具体硬件操作过程如下：

1. 根据指令的内容确定应该使用的段寄存器，比如取内存指令的内存地址所对应的数据段寄存器为DS；
2. 根据段寄存器DS的值作为选择子，以此选择子值为索引，在段描述符表（可理解为一个大数组）找到索引指向的段描述符（可理解为数组中的元素）；
3. 在段描述符中取出基地址域（段的起始地址）和地址范围域（段的长度）的值；
4. 将指令内容确定的地址偏移，与地址范围域的值比较，确保地址偏移小于地址范围，这样是为了确保地址范围不会跨出段的范围；（第一层保护）
5. 根据指令的性质（当前指令的CS值的低两位）确定当前指令的特权级，需要高于当前指令访问的数据段的特权级；（第二层保护）；
6. 根据指令的性质(指令是做读还是写操作)，需要当前指令访问的数据段可读或可写；（第三层保护）
7. 将DS指向的段描述符中基地址域的值加上指令内容中指定的访存地址段内偏移值，形成实际的物理地址（实现地址转换），发到数据地址总线上，到物理内存中寻址，并取回该地址对应的数据内容。 分段机制涉及4个关键内容：逻辑地址（Logical Address,应用程序员看到的地址，在操作系统原理上称为虚拟地址，以后提到虚拟地址就是指逻辑地址）、物理地址（Physical Address, 实际的物理内存地址）、段描述符表（包含多个段描述符的“数组”）、段描述符（描述段的属性，及段描述符表这个“数组”中的“数组元素”）、段选择子（即段寄存器中的值，用于定位段描述符表中段描述符表项的索引）。虚拟地址到物理地址的转换主要分以下两步：
8. 分段地址转换：CPU把虚拟地址（由段选择子selector和段偏移offset组成）中的段选择子值作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存的段基址加上段偏移值，形成线性地址（Linear Address，在操作系统原理上没有直接对应的描述，在没有启动分页机制的情况下，可认为就是物理地址；如果启动了分页机制，则可理解为第二级虚拟地址）。如果不启动分页存储管理机制，则线性地址等于物理地址。
9. 分页地址转换，这一步中把线性地址转换为物理地址。（注意：这一步是可选的，由操作系统决定是否需要。在后续试验中会涉及。） 上述转换过程对于应用程序员来说是不可见的。线性地址空间由一维的线性地址构成，在分段机制下的线性地址空间和物理地

址空间对等。线性地址32位长，线性地址空间容量为4G字节。分段机制中虚拟地址到线性地址转换转换的基本过程如下图所示。

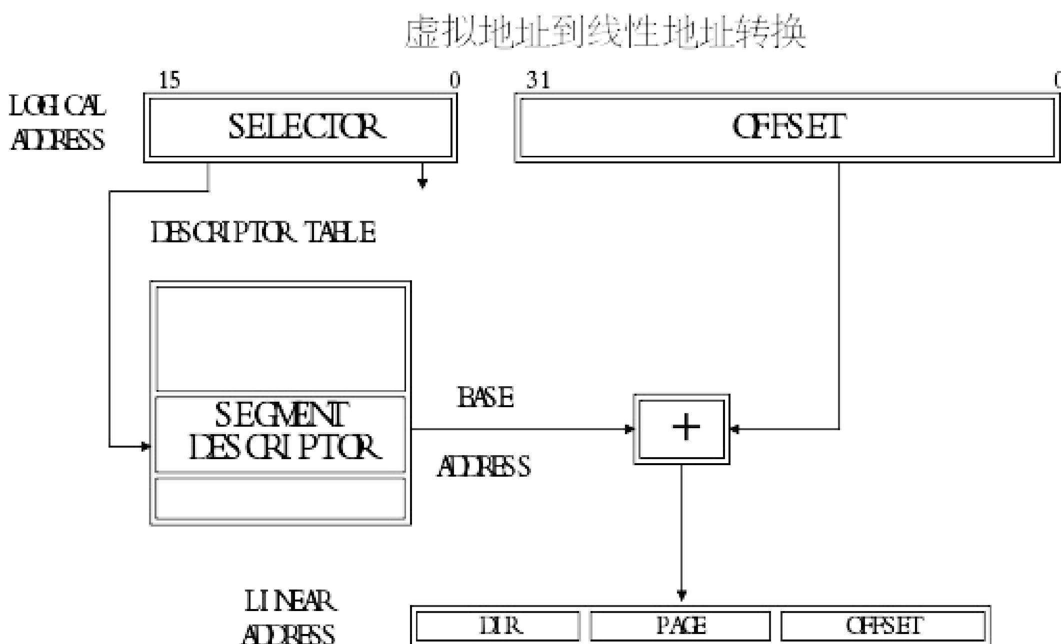


图1 分段机制中虚拟地址到线性地址转换转换基本过程

分段存储管理机制需要在启动保护模式的前提下建立。从上图可以看出，为了使得分段存储管理机制正常运行，需要在启动保护模式前建立好段描述符和段描述符表（参看bootasm.S中的“lgdt gdtdesc”语句和gdt标号/gdtdesc标号下的数据结构）。

段选择子

段选择子是用来选择哪个描述符表和在该表中索引哪一个描述符的。选择子可以做为指针变量的一部分，从而对应用程序员是可见的，但是一般是由编译器（gcc）和链接工具（ld）来设置的。段选择子的内容一般放在段寄存器中。选择子的格式如下图所示：

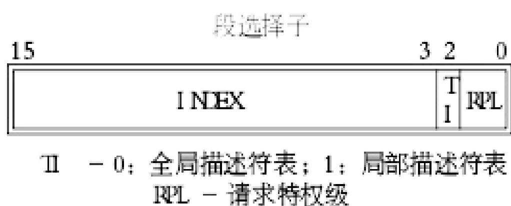


图2 段选择子结构

索引（**Index**）：在描述符表中从8192个描述符中选择一个描述符。处理器自动将这个索引值乘以8（描述符的长度），再加上描述符表的基址来索引描述符表，从而选出一个合适的描述符。

表指示位（**Table Indicator，TI**）：选择应该访问哪一个描述符表。0代表应该访问全局描述符表（GDT）；1代表应该访问局部描述符表（LDT）。LDT在实验中没有涉及。

请求特权级（**Requested Privilege Level**，**RPL**）：用于段级的保护机制，比如，段选择子是CS，则这两位表示当前执行指令的处理器所处的特权级的值，从而你可以了解到当前处理器是处于用户态（Ring 3）还是内核态（Ring 0）。在后续试验中会进一步讲解。

段描述符

在分段存储管理机制的保护模式下，每个段由如下三个参数进行定义：段基地址(Base Address)、段界限(Limit)和段属性(Attributes)。

段基地址：即线性地址空间中段的起始地址。在80386保护模式下，段基地址长32位。因为基地址长度与寻址地址的长度相同，所以任何一个段都可以从32位线性地址空间中的任何一个字节开始，而不象实方式下规定的边界必须被16整除。 在实验中，一般都简化了段机制的使用，把所有段的段基地址设置为0。

段界限：规定段的大小。在80386保护模式下，段界限用20位表示，而且段界限可以是以单字节为最小单位或以4K字节为最小单位。在实验中，一般都简化了段机制的使用，把所有段的段界限设置为0xFFFFF，以4K字节为最小单位，即段的界限为4GB；

类型（**TYPE**）：用于区别不同类型的描述符。可表示所描述的段是代码段还是数据段，所描述的段是否可读/写/执行，段的扩展方向等。 描述符特权级（**Descriptor Privilege Level**）（**DPL**）：用来实现保护机制。 段存在位（**Segment-Present bit**）：如果这一位为0，则此描述符为非法的，不能被用来实现地址转换。如果一个非法描述符被加载进一个段寄存器，处理器会立即产生异常。图2显示了当存在位为0时，描述符的格式。操作系统可以任意的使用被标识为可用（AVAILABLE）的位。 已访问位（**Accessed bit**）：当处理器访问该段（当一个指向该段描述符的选择子被加载进一个段寄存器）时，将自动设置访问位。操作系统可清除该位。 上述表示段的属性的参数通过段描述符(Segment Descriptor)来表示，一个段描述符占8字节。段描述符的结构如下图所示：

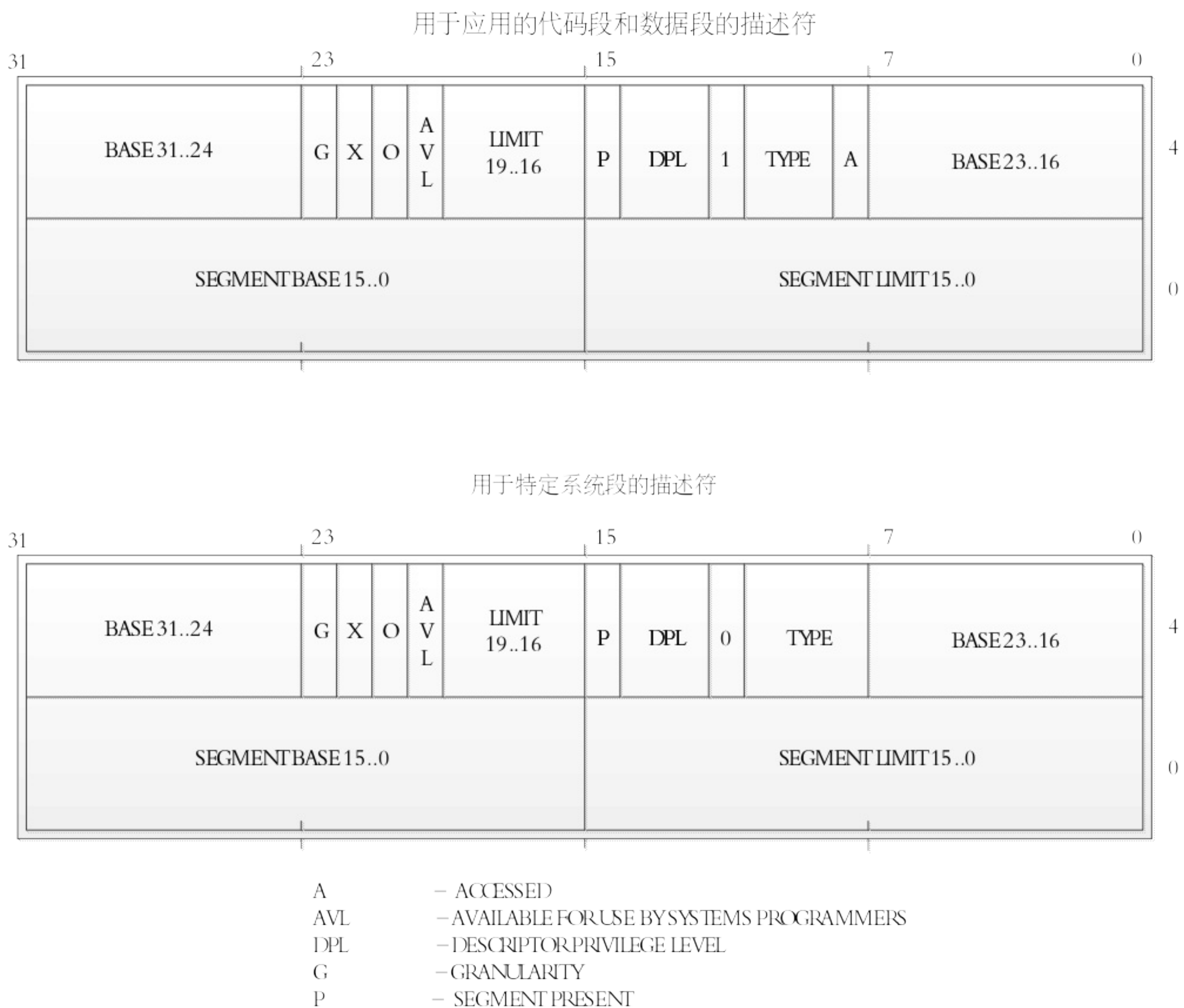


图2 段描述符结构

全局描述符表

全局描述符表的是一个保存多个段描述符的“数组”，其起始地址保存在全局描述符表寄存器GDTR中。GDTR长48位，其中高32位为基地址，低16位为段界限。由于GDT 不能用GDT本身之内的描述符进行描述定义，所以采用GDTR寄存器来表示GDT这一特殊的系统段。注意，全部描述符表中第一个段描述符设定为空段描述符。GDTR中的段界限以字节为单位。对于含有N个描述符的描述符表的段描述符实际所占空间通常可设为8*N*，若起始地址为*gdt_base*，则结束地址为*gdt_base*+*8*N-1。可参考proj1中的bootasm.S中的gdt标号和gdtdesc标号下的内容，以及lgdt指令的操作数。 全局描述符表的第一项是不能被CPU使用，所以当一个段选择子的索引（Index）部分和表指示位（Table Indicator）都为0的时（即段选择子指向全局描述符表的第一项时），可以当做一个空的选择子。当一个段寄存器被加载一个空选择子时，处理器并不会产生一个异常。但是，当用一个空选择子去访问内存时，则会产生异常。在proj1的实验中，值设置了三个段描述符，即NULL段、TEXT段和DATA段（都是4GB的访问范围）。

分段机制的系统寄存器

80386 有4个寄存器来寻址描述发表等系统数据结构，用来实现段式内存管理。内存管理寄存器包括：

- 全局描述符表寄存器 (Global Descriptor Table Register，GDTR)：指向全局段描述符表GDT
- 局部描述符表寄存器 (Local Descriptor Table Register，LDTR)：指向局部段描述符表LDT （目前用不上）
- 中断门描述符表寄存器 (Interrupt Descriptor Table Register，IDTR)：指向一张包含中断处理子程序入口点的表（IDT）
- 任务寄存器 (Task Register，TR)：这个寄存器指向当前任务信息存放处，这些信息是处理器进行任务切换所需要的。（目前用不上） 80386有四个32位的控制寄存器，分别命名位CR0、CR1、CR2和CR3。CR0包含指示处理器工作方式、启用和禁止分页管理机制、控制浮点协处理器操作的控制位。具体描述如下：
- PE（保护模式允许 Protection Enable，比特位 0）：设置PE 将让处理器工作在保护模式下。复位PE将返回到实模式工作。
- PG（分页允许 Paging， 比特位 31）： PG 指明处理器是否通过页表来转换线性地址到物理地址。在后续试验中将讲述如何设置PG位。 CR0中的位5~位30是保留位，这些位的值必须为0。CR2及CR3由分页管理机制使用，将在后续试验中讲述。在80386中不能使用CR1，否则会引起无效指令操作异常。

【实现】实模式到保护模式的切换

BIOS把bootloader从硬盘（即是我们刚才生成的ucore.img）的第一个扇区（即是我们刚才生成的bootblock）读出来并拷贝到内存一个特定的地址0x7c00处，然后BIOS会跳转到那个地址（（即CS=0，EIP=0x7c00））继续执行。至此BIOS的初始化工作做完了，进一步的工作交给了ucore的bootloader。

bootloader从哪里开始执行呢？我们【实验2-2 编译运行bootloader】中描述make工作过程的第五步就是生成了一个bootblock.asm，它的前面几行是：

```
obj/bootblock.o:     file format elf32-i386
Disassembly of section .text:
00007c00 <start>:
.set CR0_PE_ON,      0x1         # protected mode enable flag
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  7c00:    fa                    cli
```

上述代码片段指出了bootblock（即bootloader）在0x7c00虚拟地址（在这里虚拟地址=线性地址=物理地址）处的指令为“cli”，如果读者再回头看看bootasm.S中的12~15行：

```
.globl start
start:
  .code16                     # Assemble for 16-bit mode
  cli                         # Disable interrupts
  cld                         # String operations increment
```

就可以发现二者是完全一致的。而这个虚拟地址的设定是通过链接器ld完成的，我们【实验2-2 编译运行bootloader】中描述make工作过程的第四步： i386-elf-ld -N -e start -Ttext 0x7C00 -o obj/bootblock.o obj/bootasm.o obj/bootmain.o

其中“-e start”指出了bootblock的入口地址为start，而“-Ttext 0x7C00”指出了代码段的起始地址为0x7c00，这也就导致start位置的虚拟地址为0x7c00。

从0x7c00开始，bootloader用了21条汇编指令完成了初始化和切换到保护模式的工作。其具体步骤如下：

1. 关中断，并清除方向标志，即将DF置“0”，这样(E)SI及(E)DI的修改为增量。

```
cli      # Disable interrupts
cld      # String operations increment
```

2. 清零各数据段寄存器：DS、ES、FS

```
  xorw    %ax,%ax             # Segment number zero
  movw    %ax,%ds             # -> Data Segment
  movw    %ax,%es             # -> Extra Segment
  movw    %ax,%ss             # -> Stack Segment
```

3. 使能A20地址线，这样80386就可以突破1MB访存现在，而可访问4GB的32位地址空间了。可回顾2.2.1节的【历史：A20地址线与处理器向下兼容】。

```
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb    $0x2,%al
  jnz     seta20.1
  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64
seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2
  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

4. 建立全局描述符表（可回顾2.2.3节对全局描述符表的介绍），使能80386的保护模式（可回顾2.2.4节对CR0寄存器的介绍）。lgdt指令把gdt表的起始地址和界限（gdt的大小-1）装入GDTR寄存器中。而指令“movl %eax，%cr0”把保护模式开启位置为1，这时已经做好进入80386保护模式的准备，但还没有进入80386保护模式

```
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE_ON, %eax
  movl    %eax, %cr0
```

gdtdesc指出了全局描述符表（可以看成是段描述符组成的一个数组）的起始位置在gdt符号处，而gdt符号处放置了三个段描述符的信息

```
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)        # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)              # data seg
```

每个段描述符占8个字节，第一个是NULL段描述符，没有意义，表示全局描述符表的开始，紧接着是代码段描述符（位于全局描述符表的0x8处的位置），具有可读（STA_R）和可执行（STA_X）的属性，并且段起始地址为0，段大小为4GB；接下来是数据段描述符（位于全局描述符表的0x10处的位置），具有可读（STA_R）和可写（STA_W）的属性，并且段起始地址为0，段大小为4GB。

5. 通过长跳转指令进入保护模式。80386在执行长跳转指令时，会重新加载$PROT_MODE_CSEG的值（即0x8）到CS中，同时把$protcseg的值赋给EIP，这样80386就会把CS的值作为全局描述符表的索引来找到对应的代码段描述符，设定当前的EIP为0x7c32(即protcseg标号所在的段内偏移)， 根据2.2.3节描述的分段机制中虚拟地址到线性地址转换转换的基本过程，可以知道线性地址（即物理地址）为：

```
gdt[CS].base_addr+EIP=0x0+0x7c32=0x7c32
ljmp    $PROT_MODE_CSEG, $protcseg
```

6. 执行完上面的这条汇编语句后，bootloader让80386从实模式进入了保护模式。由于在访问数据或栈时需要用DS/ES/FS/GS和SS段寄存器作为全局描述符表的下标来找到相应的段描述符，所以还需要对DS/ES/FS/GS和SS段寄存器进行初始化，使它们都指向位于0x10处的段描述符（即gdt中的数据段描述符）。

```
    movw    $PROT_MODE_DSEG, %ax    # Our data segment selector
    movw    %ax, %ds                # -> DS: Data Segment
    movw    %ax, %es                # -> ES: Extra Segment
    movw    %ax, %fs                # -> FS
    movw    %ax, %gs                # -> GS
    movw    %ax, %ss                # -> SS: Stack Segment
```

在保护模式下，所有的内存寻址将经过分段机制的存储管理来完成，即每个虚拟地址访问将经过分段机制转换成线性地址，由于这时还没有启动分页模式，所以线性地址就是物理地址。

【实现】设置栈

只有设置好的合适大小和地址的栈内存空间（简称栈空间），才能有效地进行函数调用。这里为了减少汇编代码量，我们就通过C代码来完成显示。由于需要调用C语言的函数，所以需要自己建立好栈空间。设置栈的代码如下：

```
movl    $start, %esp
```

由于start位置（0x7c00）前的地址空间没有用到，所以可以用来作为bootloader的栈，需要注意栈是向下长的，所以不会破坏start位置后面的代码。在后面的小节还会对栈进行更加深入的讲解。我们可以通过用gdb调试bootloader来进一步观察栈的变化：

【实验】用**gdb**调试**bootloader**观察栈信息

1. 开两个窗口；在一个窗口中，在proj1目录下执行命令make；
2. 在proj1目录下执行 “qemu -hda bin/ucore.img -S -s”,这时会启动一个qemu窗口界面，处于暂停状态，等待gdb链接；
3. 在另外一个窗口中，在proj1目录下执行命令 gdb obj/bootblock.o；
4. 在gdb的提示符下执行如下命令，会有一定的输出：

    ```
    (gdb) target remote :1234   #与qemu建立远程链接
    (gdb) break bootasm.S:68    #在bootasm.S的第68行“movl $start, %esp”设置一个断点
    (gdb) continue              #让qemu继续执行
    ```

    这时qemu会继续执行，但执行到bootasm.S的第68行时会暂停，等待gdb的控制。这时可以在gdb中继续输入如下命令来分析栈的变化：

```
    (gdb) info registers esp
    esp            0xffd6   0xffd6    #没有执行第68行代码前的esp值
    (gdb) si                          #执行第68行代码
    69        call bootmain
    (gdb) info registers esp
    esp            0x7c00   0x7c00   #当前的esp值，即栈顶
    (gdb) si
    bootmain () at boot/bootmain.c:87    #执行call汇编指令
    87      bootmain(void) {
    (gdb) info registers esp
    esp            0x7bfc   0x7bfc    #当前的esp值0x7bfc, 0x7bfc处存放了bootmain函数
的返回地址0x7c4a，这可以通过下面两个命令了解
    (gdb) x /4x 0x7bfc
    0x7bfc: 0x00007c4a      0xc031fcfa      0xc08ed88e      0x64e4d08e
    (gdb) x /4i 0x7c40
       0x7c40 <protcseg+14>:        mov    $0x7c00,%esp
       0x7c45 <protcseg+19>:        call   0x7c6c <bootmain>
       0x7c4a <spin>:       jmp    0x7c4a <spin>
       0x7c4c <gdt>:        add    %al,(%eax)
```

【提示】

在proj1中执行

```
make debug
```

则自动完成上述大部分前期工作，即qemu和gdb的加载，且gdb会自动建立于qemu的联接并设置好断点。具体实现可参看proj1的Makefile中于debug相关的内容和tools/gdbinit中的内容。

【实现】显示字符串

bootloader只在CPU和内存中打转无法让读者很容易知道bootloader的工作是否正常，为此在成功完成了保护模式的转换后，就需要通过显示字符串来展示一下自己了。bootloader设置好栈后，就可以调用bootmain函数显示字符串了。在proj1中使用了显示器和并口两种外设来显示字符串，主要的代码集中在bootmain.c中。

这里采用的是很简单的基于Programmed I/O （PIO）方式，PIO方式是一种通过CPU执行I/O端口指令来进行数据读写的数据交换模式，被广泛应用于硬盘、光驱等设备的基础传输模式中。这种I/O访问方式使用CPU I/O端口指令来传送所有的命令、状态和数据，需要CPU全程参与，效率较低，但编程很简单。后面讲到的中断方式将更加高效。 在bootmain.c中的lpt_putc函数完成了并口输出字符的工作。输出一个字符的流程（可参看bootmain.c中的lpc_putc函数实现）大致如下：

1. 读I/O端口地址0x379，等待并口准备好；
2. 向I/O端口地址0x378发出要输出的字符；
3. 向I/O端口地址0x37A发出控制命令，让并口处理要输出的字符。

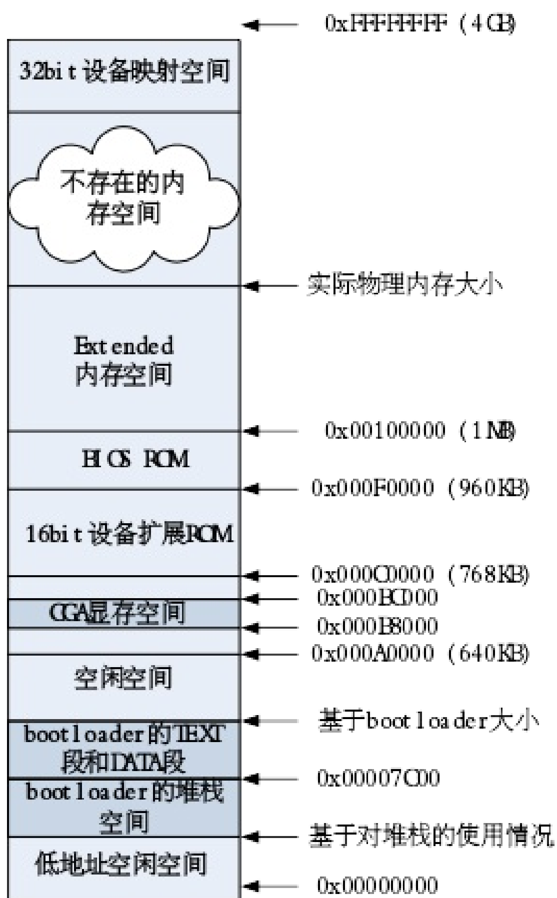
在bootmain.c中的serial_putc函数完成了串口输出字符的工作。输出一个字符的流程（可参看bootmain.c中的serial_putc函数实现）大致如下：

1. 读I/O端口地址(0x3f8+5)获得LSR寄存器的值，等待串口输出准备好；
2. 向I/O端口地址0x3f8发出要输出的字符；

在bootmain.c中的cga_putc函数完成了CGA字符方式在某位置输出字符的工作。输出一个字符的流程（可参看bootmain.c中的cga_putc函数实现）大致如下：

1. 写I/O端口地址0x3d4，读I/O端口地址0x3d5，获得当前光标位置；
2. 在光标的下一位置的显存地址空间上写字符，格式是黑色背景/白色字符;
3. 设置当前光标位置为下一位置。

proj1启动后的PC机内存布局如下图所示：



自此，我们了解了一个小巧的bootloader的实现过程，但这还仅仅是百尺竿头的第一步，它还只能显示字符串，不能加载操作系统。我们还需要扩展bootloader的功能，让它能够加载操作系统。

可读ELF格式文件的baby bootloader

实验目标

接下来，我们需要完成一个能够读取位于硬盘中OS的代码内容并加载运行OS的bootloader，这需要bootloader能够读取硬盘扇区中的数据。由于OS采用ELF执行文件格式，所以bootloader能够解析ELF格式文件，把其中的代码和数据放到内存中正确的位置。Bootloader虽然增加了这么多功能，但整个bootloader的大小还是必须小于512个字节，这样才能放到只有512字节大小的硬盘主引导扇区中。

> ucore内核不一定非要是ELF格式，基于binary格式的ucore内核也可以被bootloader识别与加载。

通过分析和实现这个bootloader，读者对设备管理的方式会有更加深入的理解，掌握bootloader/操作系统等底层系统软件是如何在保护模式下通过PIO（Programming I/O，可编程I/O）方式访问块设备硬盘；理解如何在保护模式下解析并加载一个简单的ELF执行文件。

proj2/3概述

实现描述

proj2基于proj1的主要实现一个可读硬盘并可分析ELF执行文件格式的bootloader，由于bootloader要放在512字节大小的主引导扇区中，所以不得不去掉部分显示输出的功能，确保整个bootloader的大小小于510个字节（最后两个字节用于硬盘主引导扇区标识，即“55AA”）。proj3在proj2的基础上增加了一个只能显示字符的第一代幼稚型操作系统ucore，用来验证proj2实现的bootloader能够正确从硬盘读出ucore并加载到正确的内存位置，并能把CPU控制权交给ucore。ucore在获得CPU控制权后，能够在保护模式下显示一个字符串，表明自己能够正常工作了

项目组成

这里我们分了两个project来完成此事。proj2是一个可分析ELF执行文件格式的例子，proj2整体目录结构如下所示：

```
proj2/
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- libs
|   |-- elf.h
|   |-- types.h
|   `-- x86.h
|-- Makefile
……
```

proj2与proj1类似，只是增加了libs/elf.h文件，并且bootmain.c中增加了对ELF执行文件的简单解析功能和读磁盘功能。

proj3建立在proj2基础之上，增加了一个只能显示字符的ucore操作系统，让bootloader能够把这个操作系统从硬盘上读到内存中，并跳转到ucore的起始处执行ucore的功能。proj3整体目录结构如下所示：

```
proj3
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- kern
|   |-- driver
|   |   |-- console.c
|   |   `-- console.h
|   |-- init
|   |   `-- init.c
|   `-- libs
|       `-- stdio.c
|-- libs
|   |-- elf.h
|   |-- error.h
|   |-- printfmt.c
|   |-- stdarg.h
|   |-- stdio.h
|   |-- string.c
|   |-- string.h
|   |-- types.h
|   `-- x86.h
|-- Makefile
……
```

proj3相对于proj2增加了ucore相关的文件，下面简要说明一下：

- libs目录下的printfmt.c：完成类似C语言的printf中的格式化处理；

- libs目录下的string.c：完成类似C语言的str***相关的字符串处理函数；
- libs目录下的st*.h：是支持上述两个库函数（可被内核和用户应用共享）的.h文件；
- kern/init目录下的init.c：完成ucore的初始化工作；
- kern/driver目录下的console.c：提供并口/串口/CGA方式的字符输出的console驱动；
- kern/libs/stdio.c：提供内核方式下的的cprintf函数功能；

编译运行

那接下来是如何生成一个包含了bootloader和ucore操作系统的硬盘镜像呢？我们先修改proj3目录下的Makefile，在其第五行

```
        V       := @
```

的最前面增加一个“#”（目的是让make工具程序详细显示整个project的编译过程），这样就把这行给注释了。然后在proj3目录下执行make，可以看到：

```
        ……
        ld -m    elf_i386 -Ttext 0x100000 -e kern_init -o bin/kernel obj/kern/init/ini
t.o obj/kern/libs/printf.o obj/kern/driver/console.o obj/libs/printfmt.o obj/libs/stri
ng.o
        ……
        dd if=bin/kernel of=bin/ucore.img seek=1 conv=notrunc
```

这两步是生成ucore的关键。第一步把ucore涉及的各个.o目标文件链接起来，并在bin目录下形成ELF文件格式的文件kernel，这就是我们第一个ucore操作系统，而且设定ucore的执行入口地址在0x10000，即kern_init函数的起始位置。这也就意味着bootloader需要把读出的kernel文件的代码段+数据段放置在0x10000起始的内存空间。第二步是把bin目录下的kernel文件直接覆盖到ucore.img（虚拟硬盘的文件）的bootloader所处扇区（即第一个扇区，主引导扇区）之后的扇区（第二个扇区）。如果一个扇区大小为512字节，这kernel覆盖的扇区数为上取整（kernel的大小/512字节）。

编译后运行proj3的示意图如下所示：

```
![qemu_cha1](figures/qemu_cha2.jpg)
```

【背景】访问硬盘数据控制

bootloader让80386处理器进入保护模式后，下一步的工作就是从硬盘上加载并运行OS。考虑到实现的简单性，bootloader的访问硬盘都是LBA模式的PIO（Program IO）方式，即所有的I/O操作是通过CPU访问硬盘的I/O地址寄存器完成。

一般主板有2个IDE通道（是硬盘的I/O控制器），每个通道可以接2个IDE硬盘。第一个IDE通道通过访问I/O地址0x1f0-0x1f7来实现，第二个IDE通道通过访问0x170-0x17f实现。每个通道的主从盘的选择通过第6个I/O偏移地址寄存器来设置。具体参数见下表。

```
I/O地址      功能
0x1f0     读数据，当0x1f7不为忙状态时，可以读。
0x1f2     要读写的扇区数，每次读写前，需要指出要读写几个扇区。
0x1f3     如果是LBA模式，就是LBA参数的0-7位
0x1f4     如果是LBA模式，就是LBA参数的8-15位
0x1f5     如果是LBA模式，就是LBA参数的16-23位
0x1f6     第0~3位：如果是LBA模式就是24-27位     第4位：为0主盘；为1从盘
第6位：为1=LBA模式；0 = CHS模式        第7位和第5位必须为1
0x1f7     状态和命令寄存器。操作时先给命令，再读取内容；如果不是忙状态就从0x1f0端口读数据
```

硬盘数据是储存到硬盘扇区中，一个扇区大小为512字节。读一个扇区的流程大致为通过outb指令访问I/O地址:0x1f2~-0x1f7来发出读扇区命令，通过in指令了解硬盘是否空闲且就绪，如果空闲且就绪，则通过inb指令读取硬盘扇区数据都内存中。可进一步参看bootmain.c中的readsect函数实现来了解通过PIO方式访问硬盘扇区的过程。

【背景】理解ELF文件格式

由于本章的project中，bootloader会访问ELF(Executable and linking format)格式的ucore，并把ucore加载到内存中。所以，在这里我们需要简单介绍一下ELF文件格式，以帮助我们理解ucore的整个编译、链接和加载的过程，特别是希望读者对ld链接器用到的链接地址（Link address）和操作系统相关的加载地址（Load address）有更清楚的了解。

ELF文件格式是Linux系统下的一种常用目标文件(object file)格式，有三种主要类型。可重定位文件(relocatable file)类型和共享目标文件(shared object file)类型在本实验中没有涉及。本实验的OS文件类型是可执行文件(executable file)类型，这种ELF文件格式类型提供程序的进程映像，加载程序的内存地址描述等。

简单地说，bootloader通过解析ELF格式的ucore，可以了解到ucore的代码段（机器码）/数据段（初始化的变量）等在文件中的位置和大小，以及应该放到内存中的位置；可了解ucore的BSS段（未初始化的变量，具体内容没有保存在文件中）的内存位置和大小。这样bootloader就可以把ucore正确地放置到内存中，便于ucore的正确执行。

这里只分析与本章相关的ELF可执行文件类型。ELF的执行文件映像如下所示：

ELF可执行映像

字段	值
e_ident	'E' 'L' 'F'
e_entry	0x8048090
e_phoff	52
e_phentsize	32
e_phnum	2

物理头

字段	值
p_type	PT_LOAD
p_offset	0
p_vaddr	0x8048000
p_filez	68532
p_memsz	68532
p_flags	PF_R, PF_X

物理头

字段	值
p_type	PT_LOAD
p_offset	68536
p_vaddr	0x8059BB8
p_filesz	2200
p_memsz	4248
p_flags	PF_R, PF_X

代码

数据

一个简单的ELF可执行文件的布局

ELF的文件头包含整个执行文件的数据结构elf header，描述了整个执行文件的组织结构。其定义在proj2/3中的elf.h文件中：

```
struct elfhdr {
    uint32_t e_magic;     // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 1=relocatable, 2=executable, 3=shared object, 4=core imag
e
    uint16_t e_machine;   // 3=x86, 4=68K, etc.
    uint32_t e_version;   // file version, always 1
    uint32_t e_entry;     // entry point if executable
    uint32_t e_phoff;     // file position of program header or 0
    uint32_t e_shoff;     // file position of section header or 0
    uint32_t e_flags;     // architecture-specific flags, usually 0
    uint16_t e_ehsize;    // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;     // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;     // number of entries in section header or 0
    uint16_t e_shstrndx;  // section number that contains section name strings
};
```

program header描述与程序执行直接相关的目标文件结构信息，用来在文件中定位各个段的映像，同时包含其他一些用来为程序创建进程映像所必需的信息。可执行文件的程序前面部分有一个program header结构的数组， 每个结构描述了一个“段”（segment）或者准备程序执行所必需的其它信息。目标文件的 “段”（segment） 包含一个或者多个 “节区”（section） ，也就是“段内容（Segment Contents）”。program header仅对于可执行文件和共享目标文件有意义。可执行目标文件在elfhdr的e_phentsize和e_phnum成员中给出其自身程序头部的大小。程序头部的数据结构如下表所示：

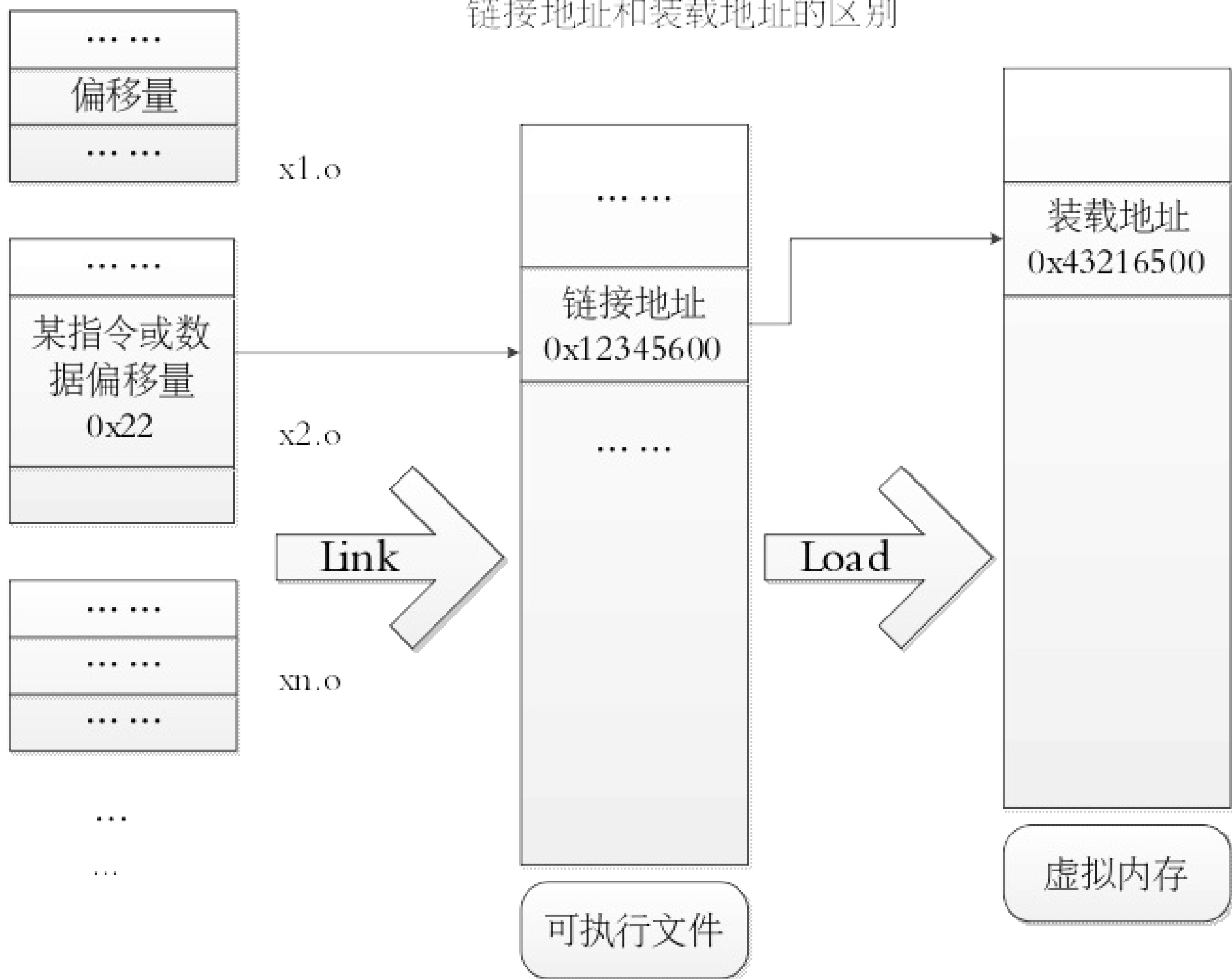
```
struct proghdr {
    uint32_t p_type;   // loadable code or data, dynamic linking info,etc.
    uint32_t p_offset; // file offset of segment
    uint32_t p_va;     // virtual address to map segment
    uint32_t p_pa;     // physical address, not used
    uint32_t p_filesz; // size of segment in file
    uint32_t p_memsz;  // size of segment in memory (bigger if contains bss）
    uint32_t p_flags;  // read/write/execute bits
    uint32_t p_align;  // required alignment, invariably hardware page size
};
```

链接地址（**Link address**）和加载地址（**Load address**）

Link Address是指编译器指定代码和数据所需要放置的内存地址，由链接器配置。Load Address是指程序被实际加载到内存的位置。一般由可执行文件结构信息和加载器可保证这两个地址相同。Link Addr和LoadAddr不同会导致：

```
直接跳转位置错误
直接内存访问(只读数据区或bss等直接地址访问)错误
堆和栈等的使用不受影响，但是可能会覆盖程序、数据区域
```

也存在Link地址和Load地址不一样的情况（如动态链接库）。在proj3中，bootloader和ucore的链接地址和加载地址是一致的。


链接地址和装载地址的区别
……
偏移量
……
x1.o
……
某指令或数据偏移量
0x22
x2.o
……
……
……
xn.o
…
…
Link
……
链接地址
0x12345600
……
可执行文件
Load
装载地址
0x43216500
虚拟内存

【背景】操作系统执行代码的组成

ucore通过gcc编译和ld链接，形成了ELF格式执行文件kernel（位于bin目录下），这样kernel的内部组成与一般的应用程序差别不大。一般而言，一个执行程序的内容是至少由 bss段、data段、text段三大部分组成。

- BSS段：BSS（Block Started by Symbol）段通常是指用来存放执行程序中未初始化的全局变量的一块存储区域。BSS段属于静态内存分配的存储空间。
- 数据段：数据段（Data Segment）通常是指用来存放执行程序中已初始化的全局变量的一块存储区域。数据段属于静态内存分配的存储空间。
- 代码段：代码段（Code Segment/Text Segment）通常是指用来存放程序执行代码的一块存储区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读,某些CPU架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

ucore和一般应用程序一样，首先是保存在像硬盘这样的非易失性存储介质上，当需要运行时，被加载到内存中。这时，需要把代码段、数据段的内容拷贝到内存中。对于位于BSS段中的未初始化的全局变量，执行程序一般认为其值为零。所以需要把BSS段对应的内存空间清零，确保执行代码的正确运行。可查看init文件中的kern_init函数的第一个执行语句“memset(edata, 0, end - edata);”。

随着ucore的执行，可能需要进行函数调用，这就需要用到栈（stack）；如果需要动态申请内存，这就需要用到堆（heap）。堆和栈是在操作系统执行过程中动态产生和变化的，并不存在于表示内核的执行文件中。栈又称堆栈， 是用户存放程序临时创建的局部变量，即函数中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用函数的栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。可以把栈看成一个寄存、交换临时数据的内存区。堆是用于存放运行中被动态分配的内存空间，它的大小并不固定，可动态扩张或缩减，这需要操作系统自己进行有效的管理。

【实现】bootloader加载并运行ucore

了解完proj2/3的组成与编译，并大致理解上述两个背景知识后，我们就可以分析bootloader加载并运行ucore操作系统的工作流程。

硬盘数据是储存到硬盘扇区中，一个扇区大小为512字节。读一个扇区的流程可参看bootmain.c中的readsect函数实现。大致如下：

1. 读I/O地址0x1f7，等待磁盘准备好；
2. 写I/O地址0x1f2~0x1f5,0x1f7，发出读取第offseet个扇区处的磁盘数据的命令；
3. 读I/O地址0x1f7，等待磁盘准备好;
4. 连续读I/O地址0x1f0，把磁盘扇区数据读到指定内存。

这个函数是被bootloader用于读取硬盘上的ucore操作系统。bootloader为了读取硬盘上的ucore操作系统，将调用bootmain函数首先读取了位于主引导扇区的后的连续8个扇区（可参见bootmain函数中的第一条语句），并把数据放到0x10000处（可回顾一下2.7.1中描述链接bin/kernel的过程），并按照数据结构elfhdr来解析这块4KB大小的数据；如果其e_magic数据域不等于ELF_MAGIC（即0x464C457F），则表示这个不是标准的ELF格式的文件；如果等于ELF_MAGIC，则继续解析，并根据其e_phnum数据域的值来读取多个program header，并根据program header的信息，了解到ucore中各个segment的起始位置和大小，然后把放在硬盘上的相关segment读入到内存中。

【实验】分析**kernel**并在**bootloader**中显示**kernel**的**segment**信息

1. 在proj3目录下执行命令make，则会在bin目录下生成kernel，即ELF执行格式文件的操作系统ucore；
2. 在proj3目录下执行命令 readelf -h bin/kernel，可得到有关elf header的如下信息

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x100000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          19872 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         17
  Section header string table index: 14
```

从中，我们可以看到kernel的入口点在0x100000，program header相对文件的偏移位置在52，elf header的大小为52字节，program header的大小为32字节。

3. 在proj3目录下执行命令 readelf -l bin/kernel，可得到有关program header的如下信息

```
Elf file type is EXEC (Executable file)
Entry point 0x100000
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x01038 0x01038 R E 0x1000
  LOAD           0x002038 0x00102038 0x00102038 0x00004 0x00004 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .data
   02
```

从中，我们可以看到kernel的入口点在0x100000，代码段位于0x100000，大小为0x1038；数据段位于0x102038，大小为0x04。

【实验】用**gdb**调试**bootloader**，并在**gdb**中显示**kernel**的**segment**信息

我们还可通过用gdb调试bootloader进行验证，具体步骤如下：

1. 开两个窗口；在一个窗口中，在proj3目录下执行命令make；
2. 在proj3目录下执行 “qemu -hda bin/ucore.img -S –s”,这时会启动一个qemu窗口界面，处于暂停状态，等待gdb链接；
3. 在另外一个窗口中，在proj3目录下执行命令 gdb obj/bootblock.o；
4. 在gdb的提示符下执行如下命令，会有一定的输出：

```
(gdb) target remote :1234   #与qemu建立远程链接
(gdb) break bootmain.c:100  #在bootmain.c的第100行设置一个断点
(gdb) continue              #让qemu继续执行
```

这时qemu会继续执行，但执行到bootmain.c的第100行时会暂停，等待gdb的控制。这时可以在gdb中继续输入如下命令来参考kernel的信息：

```
(gdb) p /x *(struct elfhdr *)0x10000  #按struct elfhdr结构显示0x10000处内容
$7 = {e_magic = 0x464c457f, e_elf = {0x1, 0x1, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0}, e_type = 0x2, e_machine = 0x3, e_version = 0x1, e_entry = 0x10000
0, e_phoff = 0x34, e_shoff = 0x4550, e_flags = 0x0, e_ehsize = 0x34, e_phentsize =
0x20,   e_phnum = 0x3, e_shentsize = 0x28, e_shnum = 0x11, e_shstrndx = 0xe}
```

查看bootmain函数，可以知道，此时在0x10000处已经读入了kernel的ELF头信息，有三个program header 表(e_phnum值),继续在gdb中敲入命令，可以得到更多信息：

```
(gdb) next                 #执行下一条指令
(gdb) p /x *ph             #获得text段的program header表信息
$5 = {p_type = 0x1, p_offset = 0x1000, p_va = 0x100000, p_pa = 0x100000, p_filesz
= 0x1038,  p_memsz = 0x1038, p_flags = 0x5, p_align = 0x1000}
(gdb) next                 #执行下一条指令
(gdb) next                 #执行下一条指令
(gdb) p /x *ph             #获得data段的program header表信息
$6 = {p_type = 0x1, p_offset = 0x2038, p_va = 0x102038, p_pa = 0x102038, p_filesz
= 0x4, p_memsz = 0x4, p_flags = 0x6, p_align = 0x1000}
```

对照readelf命令输出的信息，可以发现bootloader正确读出了text段和data段的program header表信息，并根据这些信息调用如下函数

```
-->readseg(ph->p_va, ph->p_memsz, ph->p_offset);
    -->readsect((uint8_t *)va, offset);
```

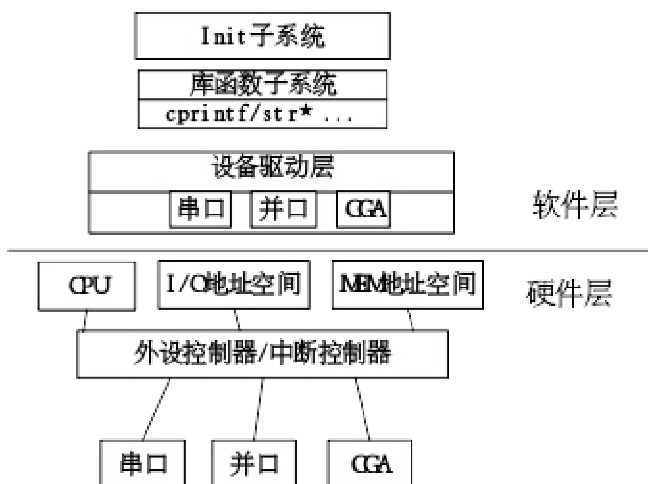
把这两个段的内容读入到正确的线性内存地址中。然后再根据e_entry = 0x100000，跳转到0x100000处去执行，这其实就是把处理器控制权转移给了ucore了。

【实现】可输出字符串的ucore

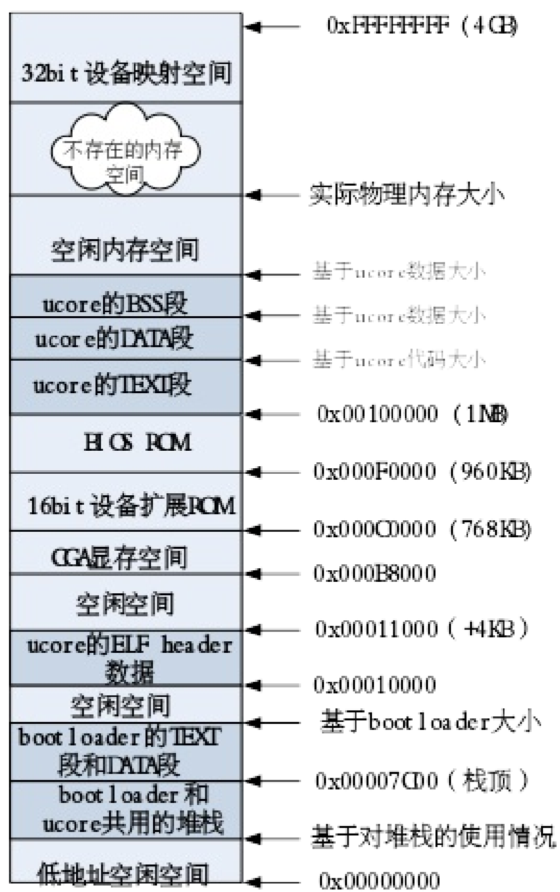
proj3包含了一个只能输出字符串的简单ucore操作系统，虽然简单，但它也体现了操作系统的一些结构和特征，比如它具有：

- 完成给ucore的BSS段清零并显示一个字符串的内核初始化子系统（init.c）
- 提供串口/并口/CGA显示的驱动程序子系统（console.c）
- 提供公共服务的操作系统函数库子系统（printf.c printfmt.c string.c）

这体现了操作系统的一个基本特征：资源管理器。从操作系统原理我们可以知道一台计算机就是一组资源，这些资源用于对数据的移动、存储和处理并进行控制。在proj3中的ucore操作系统目前只提供了对串口/并口/CGA这三种I/O设备的硬件资源的访问，每个I/O设备的操作都有自己特有的指令集或控制信号（对照一下serial_putc/lpt_putc/cga_putc函数的实现），操作系统隐藏这些细节，并提供了统一的接口（看看cprintf函数的实现），因此程序员可以使用简单的printf函数来写这些设备，达到显示数据的效果。目前操作系统的逻辑结构图架构如下图所示：



在PC中的地址空间布局图如下所示：


32bit 设备映射空间
不存在的内存空间
空闲内存空间
ucore的BSS段
ucore的DATA段
ucore的TEXT段
BIOS ROM
16bit 设备扩展ROM
CGA显存空间
空闲空间
ucore的ELF header数据
空闲空间
bootloader 的TEXT段和DATA段
bootloader 和ucore共用的堆栈
低地址空闲空间
0xFFFFFFFF（4GB)
实际物理内存大小
基于ucore数据大小
基于ucore数据大小
基于ucore代码大小
0x00100000（1MB)
0x000F0000（960KB)
0x000C0000（768KB)
0x000B8000
0x00011000（+4KB）
0x00010000
基于bootloader大小
0x00007C00（栈顶）
基于对堆栈的使用情况
0x00000000

能显示函数调用关系的ucore

操作系统中存在很多函数，通过函数间的调用来完成各种功能。在操作系统运行过程中，维持函数之间的调用关系，以及函数内部的局部变量是栈（stack）的基本功能。我们需要能够理解在操作系统中栈的实现细节和功能，这样能够更好地理解操作系统中函数如何相互调用，发现可能存在的问题。

实验目标

为了理解操作系统中的函数调用关系、传递参数和函数局部变量，我们设计了proj3.1，在ucore中增加了一个monitor子功能模块，能够分析出ucore在执行过程中的函数调用关系和函数传递的参数。通过分析proj3.1的实现，读者可了解基本栈结构，栈处理流程，GCC编译器的参数传递约定和构建函数调用关系的具体实现。

proj3.1概述

1. 实现描述
   proj3.1建立在proj3的基础上，通过增加了一个monitor功能子模块，实现了一个能显示函数调用关系的ucore。简单地说proj3.1根据GCC生成的栈构建代码、函数参数压栈约定和实际函数调用过程中的栈结构内存空间，分析并显示函数调用关系。具体完成此工作的是print_stackframe函数。

2. 项目组成
   proj3.1整体目录结构中新增加的主要内容如下所示：

```
proj3.1
|-- kern
|   |-- debug
|   |   |-- assert.h
|   |   |-- kdebug.c
|   |   |-- kdebug.h
|   |   |-- monitor.c
|   |   |-- monitor.h
|   |   |-- panic.c
|   |   `-- stab.h
|   |-- driver
|   |   `-- kbdreg.h
|   |-- init
|   |   `-- init.c
|   |-- libs
|   |   |-- readline.c
|   |   `-- stdio.c
|   `-- trap
|       `-- trap.h
|-- Makefile
`-- tools
    |-- kernel.ld
……
```

proj3.1是基于proj3进一步扩展完成的。相对于proj3，一共增加了10个文件，主要集中在debug目录下，这一个比较大的跨越。不过仔细看来主要增加和修改的文件不多，具体新增内容如下所示：

- kern/debug/monitor.[ch]：监视系统运行的monitor交互子模块
- kern/debug/debug.[ch]：实现内存地址到函数名的映射和分析显示函数调用关系；
- kern/libs/readline.c：实现monitor的接收字符输入的功能；
- kern/driver/kbdreg.h：定义了键盘的键值；
- kern/trap/trap.h：为了向后兼容中断的处理，先在此处写了一个空的trapframe结构；
- tools/kernel.ld：指导ld工具软件链接各个.o目标文件形成ucore的kernel的链接脚本；

3. 编译运行 编译并运行proj3.1的命令如下：

```
cd proj3.1
make
make qemu
```

当“K>”提示符出现后，可以敲入“help”字符串，可以看到当前的monitor有三个命令可以输入：help、kerninfo、backtrace。我们敲入“backtrace”字符串，则可以得到如下显示界面：

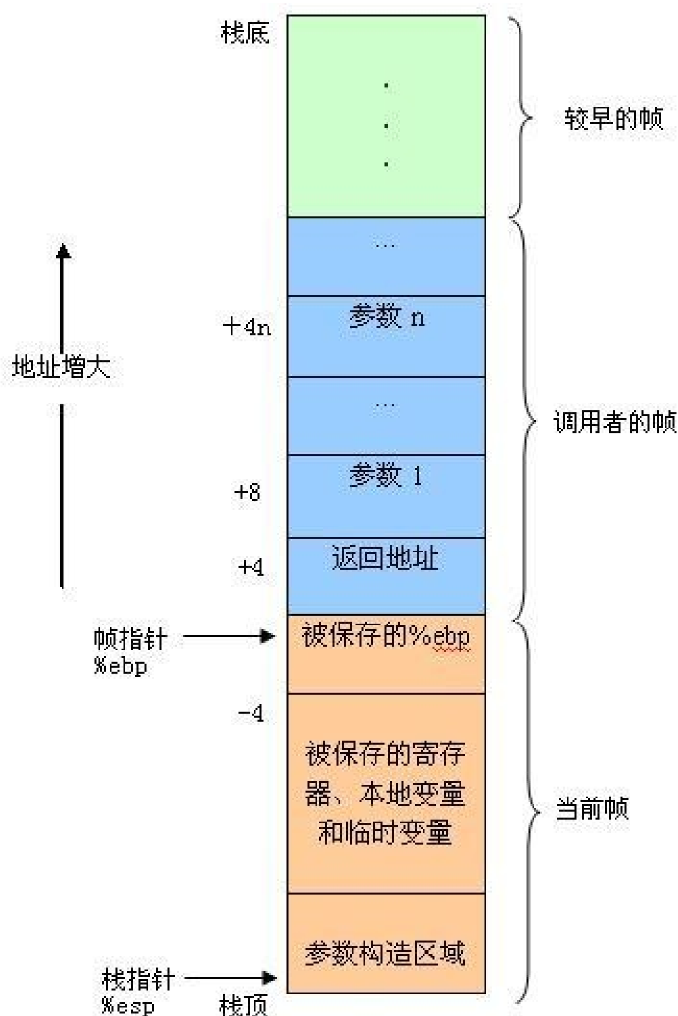
```
(THU.CST) os is loading ...

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> help
help - Display this list of commands.
kerninfo - Display information about the kernel.
backtrace - Print backtrace of stack frame.
K> backtrace
ebp:0x00007b08 eip:0x0010073a args:0x00010094 0x00000000 0x00007b88 0x00100985
    kern/debug/kdebug.c:216: print_stackframe+25
ebp:0x00007b18 eip:0x00100a76 args:0x00000000 0x00007b3c 0x00000000 0x0000000a
    kern/debug/monitor.c:94: mon_backtrace+11
ebp:0x00007b88 eip:0x00100985 args:0x00108560 0x00000000 0x00007bb8 0x0010124c
    kern/debug/monitor.c:55: runcmd+135
ebp:0x00007bb8 eip:0x001009f8 args:0x00000000 0x00101ef0 0x0000065c 0x00000000
    kern/debug/monitor.c:70: monitor+75
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000 0x00000000 0x00007c4f
    kern/init/init.c:22: kern_init+89
ebp:0x00007bf8 eip:0x00007d5b args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8
```

通过上图可以看到monitor能够把当前的函数调用关系给显示出来，而且不仅仅给出函数调用返回处的eip地址，还显示出了在实际源代码处的文件名和行号。如果ucore在执行过程中由于某种异常错误激发monitor执行（以后有这样的实验），我们就可以很容易找到问题出现在什么地方了。下面我们将从栈的基本概念、栈结构和栈处理过程等方面来分析上图中背后的东西。

【背景】栈结构和处理过程

根据数据结构课程中的描述，栈是限定仅在表尾进行插入（即入栈操作）或删除操作（即出栈操作）的线性表[严蔚敏著的《数据结构》书]。因此，栈的表尾端成为称为栈顶（stack top），表头端称为栈底（stack bottom）。在X86CPU架构中有专门的指令“push”来完成入栈操作，“pop”指令来完成出栈操作，栈顶指针寄存器ESP时刻指向栈的栈顶。比较有趣的是，在x86中，采用的是满降序栈（full descending stack）机制，即栈底在高地址，栈顶在低地址，入栈的方向是向低地址进行，出栈的方向是向高地址进行，栈指针指向上次写的最后一个数据单元。GCC编译器规定的函数栈帧（stack frame）是一块存放某函数的局部变量、参数、返回地址和其它临时变量的内存空间，栈帧的大致结构和操作如下图所示：



操作系统中使用栈的目的与一般应用程序类似，不外乎包括：支持函数调用、传递函数参数、局部变量（也称自动变量）存储、函数返回值存储、在函数内部保存可能被修改的寄存器的值供恢复。除此之外，在后续讲到的中断处理、内核态/用户态切换、进程切换等方面，也需要使用栈来保存被打断的执行所用到的寄存器等硬件信息。由于在操作系统中要完成许

多非常规的栈空间数据处理，比如直接修改保存在栈帧中的返回地址，使得函数返回到不同的地方去，所以我们需要在计算机体系结构和机器代码级别更加深入地理解操作系统是如何具体进行栈处理的。下面，我们将结合调试运行proj3.1来分析内核中的函数调用关系。

【实现】分析内核函数调用关系

首先，ucore需要建立一个空的栈空间，然后才能进行函数调用、参数传递等处理工作。ucore是在哪里建立的栈呢？其实ucore是借用了bootloader的栈空间，而bootloader在bootasm.S中的如下语句建立的栈空间：

```
# Set up the stack pointer and call into C.
movl $0x0, %ebp
movl $start, %esp
```

可以看到bootloader把栈底设置到了$start地址处，正好是bootloader的起始地址0x7c00。不过由于入栈操作中的esp是向下增长的，所以不会覆盖bootloader的内容。那ebp有何作用呢？我们先暂时放在一边，继续跟踪代码的执行。

接下来，bootloader会调用bootmain（）函数，而bootmain（）函数会在加载完ucore后，调用ucore的起始函数kern_init（）。在ucore的继续执行过程中，还将有如下的函数调用过程：

```
kern_init-->monitor-->runcmd-->mon_backtrace-->print_stackframe。
```

这通过看源代码或执行monitor中的backtrace命令都可以了解到。

我们可以结合前面的实验来说明ucore是如何分析出这样的调用关系的。

操作系统中的中断（也称异常）技术是操作系统的重要功能，是计算机硬件和软件相配合产生的重要技术。简单地说，中断处理是指由于有紧急事件产生，需要打断当前CPU的正常执行，转而处理紧急事件，处理完毕后，恢复到被打断的地方继续执行。通过中断机制，计算机系统可以高效地处理外设请求，可以快速响应应用软件的异常或请求，也可以有规律地打断应用程序的执行，把执行CPU控制权还到操作系统手中，从而使得整个计算机系统的资源可控。但单纯的操作系统原理书籍很难深入分析中断的处理细节。我们希望通过后续的proj4/4.1.1等的实验，让读者了解到ucore操作系统如何完成上述事情。

首先我们需要了解GCC生成的C函数调用过程：

1. 调用函数为了传递参数给被调用函数，需要执行0到n个push指令把函数参数入栈，然后会执行一个call指令，在call指令内部执行过程中，还把返回地址（即CALL指令下一条指令的地址）也入栈了。
2. GCC编译器会在每个函数的起始部分插入类似如下指令(可参看obj/kernel.asm文件内容)：

```
push   %ebp
mov    %esp,%ebp
sub    $NUM,%esp
```

在ucore执行到一个函数的函数体时，已经有以下数据顺序入栈：调用函数的参数，函数返回地址。由此得到类似如下的栈结构（参数入栈顺序跟调用方式有关， 这里以C语言默认的CDECL为例）：

```
高地址方向
|------------------------|
|------------------------| <-----------
|     .........       |
|   argument 3     |  Caller's stack frame
|   argument 2     |
|   argument 1     |
|   return address   |<---------- esp
|------------------------|
|     .........       |
低地址方向
```

“push %ebp”和“mov %esp,%ebp”这两条指令实在隐含了对函数调用关系链的建立：首先将ebp入栈，然后将栈顶指针esp赋值给ebp，此时的栈结构如下所示：

高地址方向 |------------------------| |------------------------<----------- | .... | | argument 3 | Caller's stack frame | argument 2 | | argument 1 | | return address |<----------- |----previous ebp-----|<------ ebp, esp |------------------------| | .... | 低地址方向

“mov %esp,%ebp”这条指令表面上看是用esp把ebp的旧值覆盖了，但在这条语句之前，ebp旧值已经被压栈（位于栈顶），而新的ebp又恰恰指向栈顶。第三条语句“sub $NUM,%esp”把esp减少了NUM个值，这其实是建立了函数的局部变量、寄存器保存的空间。此时的栈结构如下所示：

```
高地址方向
|------------------------|
|------------------------<-----------
|     .........       |
|   argument 3     |  Caller's stack frame
|   argument 2     |
|   argument 1     |
|   return address  |<-----------
|----previous ebp-----|<------ ebp, esp
|------------------------|
|     .........       |
|   saved registers  |
|     .........       | Current(Callee's) stacl frame
|  local  variables  |
|     …......      |
|----------------------- |<-------- esp
|------------------------|
|     .........       |
低地址方向
```

到此时为止，ebp寄存器处于函数调用关系链中一个非常重要的地位。ebp寄存器中存储着栈中的一个地址（栈帧分界处），此地址是“老”ebp入栈后的栈顶。那么以该该地址为基准，向高地址方向（即栈底方向）能获取返回地址、参数值，向低地址方向（栈顶方向）能获取函数局部变量值，而该地址处又存放着上一层函数调用时的ebp值。由于ebp中的地址处总是“上一层函数调用时的ebp值”，这样，就能通过把ebp的内容作为寻找上一个调用函数的栈帧的指针，如此形成递归，直至到达栈底。这就可找到整个的函数调用栈。这也是kdebug.c中print_stackframe函数的实现内容。

可管理中断并处理中断方式I/O的ucore

实验目标

前面的project都没有引入中断机制，所以bootloader和ucore都是正常地顺序执行，不会受到外界（比如外设）的“干扰”。虽然实现简单，但无法解决上述问题。我们需要扩展ucore的功能，让ucore能够支持中断，这需要读者了解基本的80386硬件中断机制，对保护模式有更深入的了解；需要清楚在中断的处理过程中，硬件主动完成了什么事情，软件在硬件完成的基础上又要完成哪些事情。通过学习和实践，读者可以了解清楚上述问题，并进一步知道通过操作系统的中断处理例程（Interrupt Process Routine, IPR）完成设备请求处理的方法等。

proj4概述

实现描述

proj4建立在proj3.1的基础上，实现了一个通过中断机制完成设备（键盘、串口和时钟）中断请求处理的ucore。简单地说proj4扩展与中断相关的工作有两个，一个是初始化中断，涉及初始化中断控制器8259A（打通外设与CPU的通路）和中断门描述符表（建立外设中断与中断服务例程的联系）和各种外设。以proj4的ucore为例，操作系统内核启动以后，kern_init函数（kern/init/init.c）通过调用pic_init函数完成对中断控制器的初始化工作，调用idt_init函数完成了对整个中断门描述符表的创建，调用cons_init和clock_init函数完成对串口、键盘和时钟外设的中断初始化工作。

ucore的另一个重要工作是中断服务，即收到中断后，对中断进行处理的中断服务例程（比如收到100个时钟中断后，显示一个字符串“100 ticks”）等。这主要集中在vectors.S（包括256个中断服务例程的入口地址和第一步初步处理实现）、trapentry.S（紧接着第一步初步处理后，进一步完成第二步初步处理的实现以及中断处理完毕后的返回准备工作）和trap.c中（紧接着第二步初步处理后，继续完成具体的各种中断处理操作）。

项目组成

proj4整体目录结构如下所示：

```
proj4
|-- kern
|   |-- driver
|   |   |-- clock.c
|   |   |-- clock.h
|   |   |-- console.c
|   |   |-- console.h
|   |   |-- picirq.c
|   |   `-- picirq.h
|   |-- init
|   |   `-- init.c
|   |-- mm
|   |   |-- memlayout.h
|   |   `-- mmu.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
`-- tools
    `-- vector.c
……
```

proj4是基于proj3.1（会在内置监控自身运行状态的ucore一节中进一步说明）进一步扩展完成的。相对于proj3.1，增加了大约10个文件，相关增加和改动主要集中在kern/driver和kern/trap目录下，使得ucore具有外设中断处理功能，这一个比较大的跨越。主要增加和修改的文件如下所示：

- tools/vector.c：生成vectors.S，此文件包含了中断向量处理的统一实现。
- kern/driver/intr.[ch]：实现了通过设置CPU的eflags来屏蔽和使能中断的函数；
- kern/driver/picirq.[ch]：实现了对中断控制器8259A的初始化和使能操作；
- kern/driver/clock.[ch]：实现了对时钟控制器8253的初始化操作；
- kern/driver/console.[ch]：实现了对串口和键盘的中断方式的处理操作；
- kern/trap/vectors.S：包括256个中断服务例程的入口地址和第一步初步处理实现；
- kern/trap/trapentry.S：紧接着第一步初步处理后，进一步完成第二步初步处理；并且有恢复中断上下文的处理，即中断处理完毕后的返回准备工作；
- kern/trap/trap.[ch]：紧接着第二步初步处理后，继续完成具体的各种中断处理操作；

编译运行

编译运行

编译并运行proj4的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

通过上图可以看到时钟中断已经能够正常相应，每隔100个时钟中断会显示一次“100 ticks”的信息。一个简单的显示信息的背后蕴藏着中断处理的复杂实现。下面我们将从中断基本概念、中断控制器、保护模式的中断处理机制等方面来分析上图中背后的东西。

【背景】理解CPU对外设中断的硬件支持

操作系统需要对计算机系统中的各种外设进行管理，这就需要CPU和外设能够相互通信才行。一般外设的速度远慢于CPU的速度。如果让操作系统通过CPU“主动关心”外设的事件，即采用通常的轮询(polling)机制，则太浪费CPU资源了。所以需要操作系统和CPU能够一起提供某种机制，让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后在恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。但中断引入的不确定性和异步性导致了设计和实现操作系统更加困难。

本章只描述保护模式下的中断处理过程。当CPU收到外设中断（可通过可编程中断控制器芯片8259A发给CPU中断信息）、CPU自身产生的故障（Fault）或CPU自身“有意”产生的陷阱（trap）时，它会暂停执行当前的程序或任务，通过一定的机制跳转到负责处理这个事件的相关处理例程中，在完成对这个事件的处理后再跳回到刚才被打断的程序或任务中。中断向量和中断服务例程的对应关系主要是由IDT（中断门描述符表）来描述。操作系统在IDT中设置好各种中断向量对应的中断描述符，而中断描述符指出了中断服务例程的起始地址，留待CPU在产生中断后查询对应中断服务例程的起始地址。而IDT本身的起始地址保存在IDTR寄存器中。

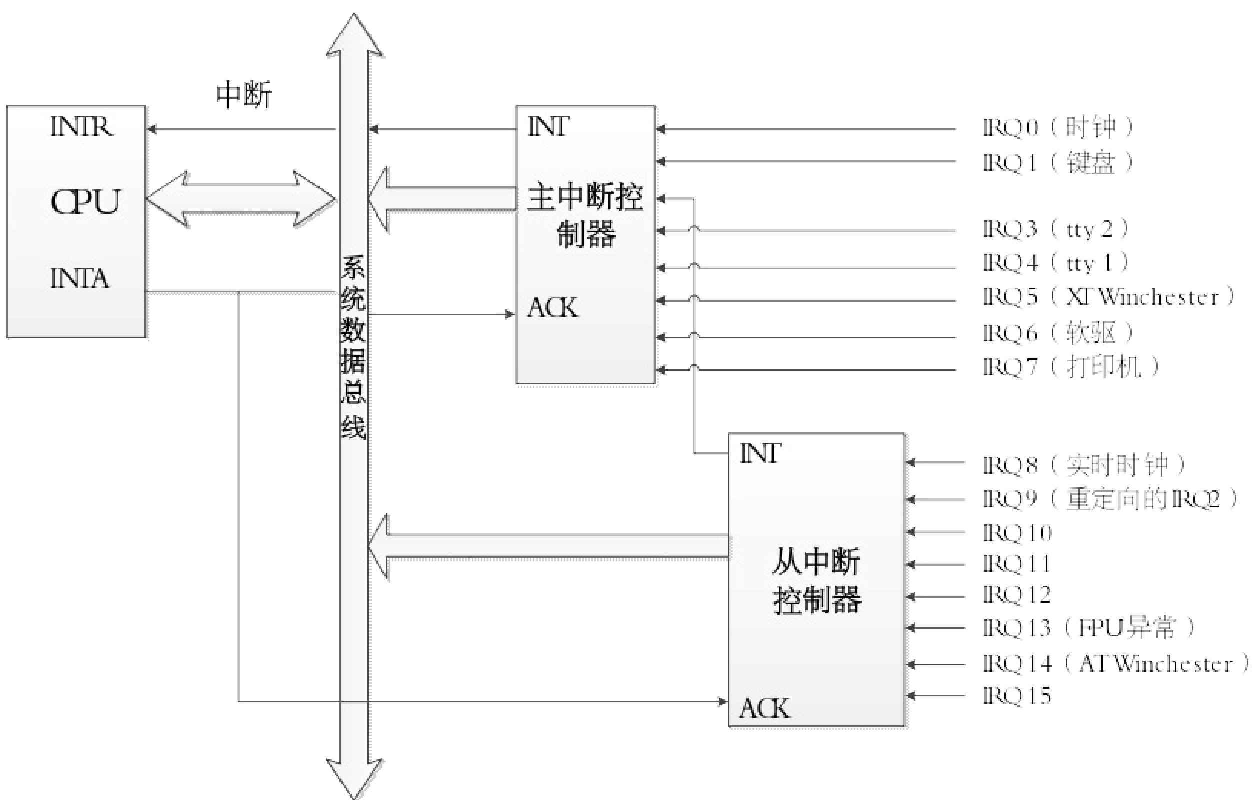
80386共支持256种中断，其中故障（Fault）和陷阱(Trap)由CPU自身产生，不使用中断控制器，也不能被屏蔽。外设中断又分为可屏蔽中断（INTR）和非屏蔽中断（NMI），I/O设备产生的中断请求（IRQ）引起可屏蔽中断，而紧急的外设事件（如掉电故障）引起的中断事件引起非屏蔽中断。

非屏蔽中断和异常的编号是固定的，而屏蔽中断的编号可以通过对中断控制器的编程来调整。256个中断的分配如下：

- 0~31号的中断对应于故障、陷阱和非屏蔽外设中断。
- 32~47号的中断分配给可屏蔽外设中断。
- 48~255号的中断可以用软件来设置。比如ucore可用其中的一个中断号来实现系统调用。

外设可屏蔽中断

80386通过两片中断控制器8259A来响应15个外中断源，每个8259A可管理8个中断源。第一级（称主片）的第二个中断请求输入端，与第二级8259A（称从片）的中断输出端INT相连，如下图所示。IRQ号和中断号之间的映射关系可以通过中断控制器来调整。



级联的 8259A架构

在中断产生过程中，中断控制器8259A监视外设产生的中断请求（IRQ）信号，如果外设产生了一个中断请求信号，则8259A执行如下操作：

1. 把接受到的IRQ信号转换成一个对应的中断编号；
2. 把这个中断编号值存放在中断控制器的一个I/O地址单元中，CPU通过数据/地址总线可访问到此I/O地址单元；
3. 给CPU的INTR引脚触发信号，即发出一个中断；
4. 等待直到CPU通过INTA引脚确认这个中断信号，清除INTR引脚上的触发信号。

屏蔽外部I/O请求有两种方法。一种是从CPU的角度清零CPU的EFLAG的中断标志位（IF）；另一种是从中断控制器的角度，即通过把中断控制器中的中断屏蔽寄存器（IMR）相应位置1，则表示禁用某条中断线。

陷阱、故障和非屏蔽中断

陷阱和故障是CPU内部执行指令的过程中产生的中断事件。非屏蔽中断就是计算机内部硬件出错时引起的紧急故障情况。80386处理器发布了大约20种陷阱、故障或非屏蔽中断。在某些故障产生时，CPU会产生一个硬件错误码并压入内核栈中。

在下表中给出了在实验中可能碰到的80386中陷阱的中断号、名称、类别及简单描述。更多的信息可以在Intel的技术文档中找到。

表 ucore中异常的简单描述

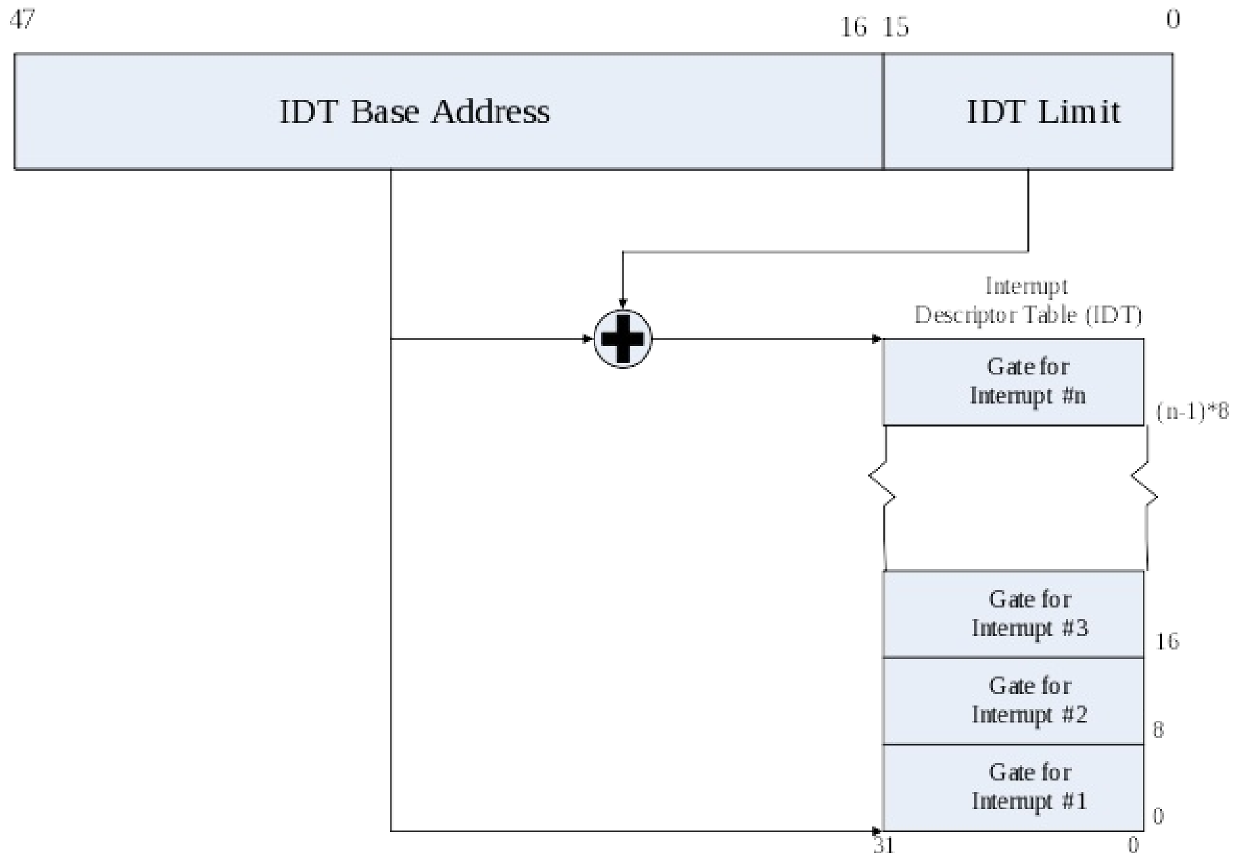
中断号	名称	类别	简单描述
8	双重故障	故障	在处理故障中又产生了故障
11	段不存在	故障	访问一个不存在的段
12	栈段异常	故障	超过栈段界限，或由ss标识的段不存在
13	通用保护	故障	违反了保护模式下的某种保护规则
14	页异常	故障	页不在内存，或违反了一种分页保护机制

中断门描述符表（Interrupt Descriptor Table）

中断门描述符表把每个中断或异常编号和一个指向中断服务例程的描述符联系起来。同GDT一样，IDT是一个8字节的描述符数组，但IDT的第一项可以包含一个描述符。CPU把中断（异常）号乘以8做为IDT的索引。IDT可以位于内存的任意位置，CPU通过IDT寄存器（IDTR）的内容来寻址IDT的起始地址。指令LIDT和SIDT用来操作IDTR。两条指令都有一个显示的操作数：一个6字节表示的内存地址。指令的含义如下：

- LIDT（Load IDT Register）指令：使用一个包含线性地址基址和界限的内存操作数来加载IDT。操作系统创建IDT时需要执行它来设定IDT的起始地址。这条指令只能在特权级0执行。
- SIDT（Store IDT Register）指令：拷贝IDTR的基址和界限部分到一个内存地址。这条指令可以在任意特权级执行。

IDT和IDTR寄存器的结构和关系如下图所示：



在保护模式下，最多会存在256个Interrupt/Exception Vectors。范围[0，31]内的32个向量被故障中断和NMI（不可屏蔽）中断使用，但当前并非所有这32个向量都已经被使用，有几个当前没有被使用。范围[32，255]内的向量被保留给用户定义的中断，可将它们用作外部I/O设备中断（8259A IRQ），或者系统调用（System Call 、Software Interrupts）等。

门描述符（**Gate Descriptors**）

在保护模式下，中断门描述符表（IDT）中的每个表项由8个字节组成，其中的每个表项叫做一个门描述符（Gate Descriptor），“门”的含义是指当中断发生时必须先访问这些“门”，能够“开门”（即将要进行的处理需通过特权检查，符合设定的权限等约束）后，然后才能进入相应的处理程序。而门描述符则描述了“门”的属性（如特权级、段内偏移量等）。在IDT中，可以包含如下3种类型的系统段描述符：

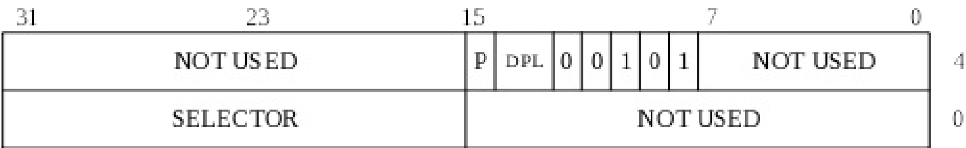
- 中断门描述符（Interrupt-gate descriptor）： 用于中断处理，其类型码为110，中断门包含了一个外设中断或故障中断的处理程序所在段的选择子和段内偏移量。当控制权通过中断门进入中断处理程序时，处理器清IF标志，即关中断，以避免嵌套中断的发生。中断门中的DPL（Descriptor Privilege Level）为0，因此用户态的进程不能访问中断门。所有的中断处理程序都由中断门激活，并全部限制在内核态。
- 陷阱门描述符（Trap-gate descriptor）：用于系统调用，其类型码为111，与中断门类似，其唯一的区别是，控制权通过陷阱门进入处理程序时维持IF标志位不变，也就是说，

不关中断。

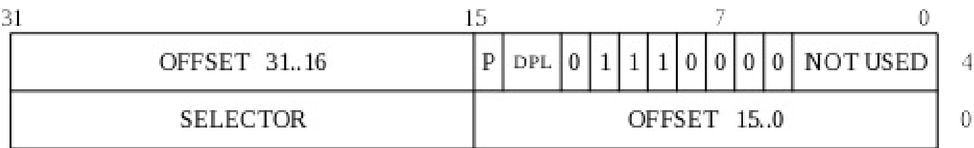
- 任务门描述符（Task-gate descriptor）和调用门描述符（Call-gate descriptor）：这两种主要是Intel设置的“任务”切换的手段，在本书中暂时没有使用。

下图图显示了80386的中断门描述符、陷阱门描述符的格式：

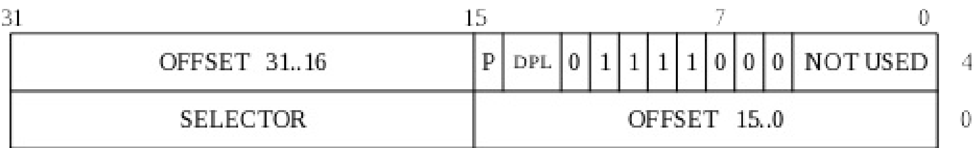
80386 TASK GATE



80386 INTERRUPT GATE



80386 TRAP GATE



中断处理中硬件负责完成的工作

中断服务例程包括具体负责处理中断（异常）的代码是操作系统的重要组成部分。需要注意区别的是，有两个过程由硬件完成：

- 硬件中断处理过程1（起始）：从CPU收到中断事件后，打断当前程序或任务的执行，根据某种机制跳转到中断服务例程去执行的过程。其具体流程如下：

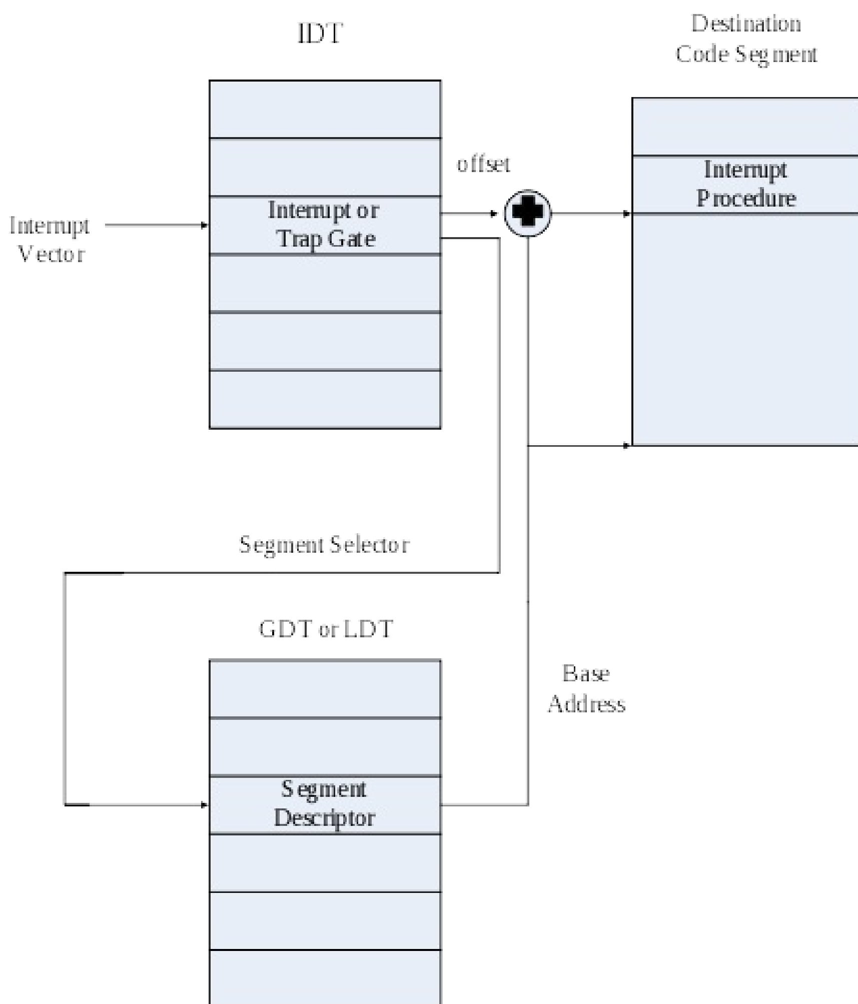
  1. CPU在执行完当前程序的每一条指令后，都会去确认在执行刚才的指令过程中中断控制器（如8259A）是否发送中断请求过来，如果有那么CPU就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量；
  2. CPU根据得到的中断向量（以此为索引）到IDT中找到该向量对应的中断描述符，中断描述符里保存着中断服务例程的段选择子；
  3. CPU使用IDT查到的中断服务例程的段选择子从GDT中取得相应的段描述符，段描述符里保存了中断服务例程的段基址和属性信息，段描述符的基址+中断描述符中的偏移地址形成了中断服务例程的起始地址；

4. CPU会根据CPL和中断服务例程的段描述符的DPL信息确认是否发生了特权级的转换。比如当前应用程序正运行在用户态，而中断服务例程是运行在内核态的，则意味着发生了特权级的转换，这时CPU会从当前应用程序的TSS信息（该信息在内存中的起始地址存在TR寄存器中）里取得该程序的内核栈地址，即包括内核态的ss和esp的值，并立即将系统当前使用的栈切换成新的内核栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的用户态的ss和esp压到新的内核栈中保存起来；如果当前程序运行在内核态，则不会发生特权转移
5. CPU需要开始保存当前被打断的用户态程序的现场（即一些寄存器的值），以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode（如果是有错误码的异常）信息；
6. CPU把中断服务例程的地址加载到cs和eip寄存器中，开始执行中断服务例程。这意味着先前的程序被暂停执行，中断服务程序正式开始工作。

- 硬件中断处理过程2（结束）：每个中断服务例程在有中断处理工作完成后需要通过iret（或iretd）指令恢复被打断的程序的执行。CPU执行IRET指令的具体过程如下：

1. 程序执行这条iret指令时，首先会从内核栈里弹出先前保存的被打断的程序的现场信息，即eflags，cs，eip重新开始执行；
2. 如果存在特权级转换（从内核态转换到用户态），则还需要从内核栈中弹出用户态栈的ss和esp，这样也意味着栈也被切换回原先使用的用户态的栈了；
3. 如果此次处理的是带有错误码（errorCode）的异常，CPU在恢复先前程序的现场时，并不会弹出errorCode。这一步需要通过软件完成，即要求相关的中断服务例程在调用iret返回之前添加出栈代码主动弹出errorCode。

下图显示了从中断向量到GDT中相应中断服务程序起始位置的定位方式：



中断处理的特权级转换

中断处理得特权级转换是通过门描述符（gate descriptor）和相关指令来完成的。一个门描述符就是一个系统类型的段描述符，一共有4个子类型：调用门描述符（call-gate descriptor），中断门描述符（interrupt-gate descriptor），陷阱门描述符（trap-gate descriptor）和任务门描述符（task-gate descriptor）。与中断处理相关的是中断门描述符和陷阱门描述符。这些门描述符被存储在中断门描述符表（Interrupt Descriptor Table，简称IDT）当中。CPU把中断向量作为IDT表项的索引，用来指出当中断发生时使用哪一个门描述符来处理中断。中断门描述符和陷阱门描述符几乎是一样的。中断发生时实施特权检查的过程如下图所示：

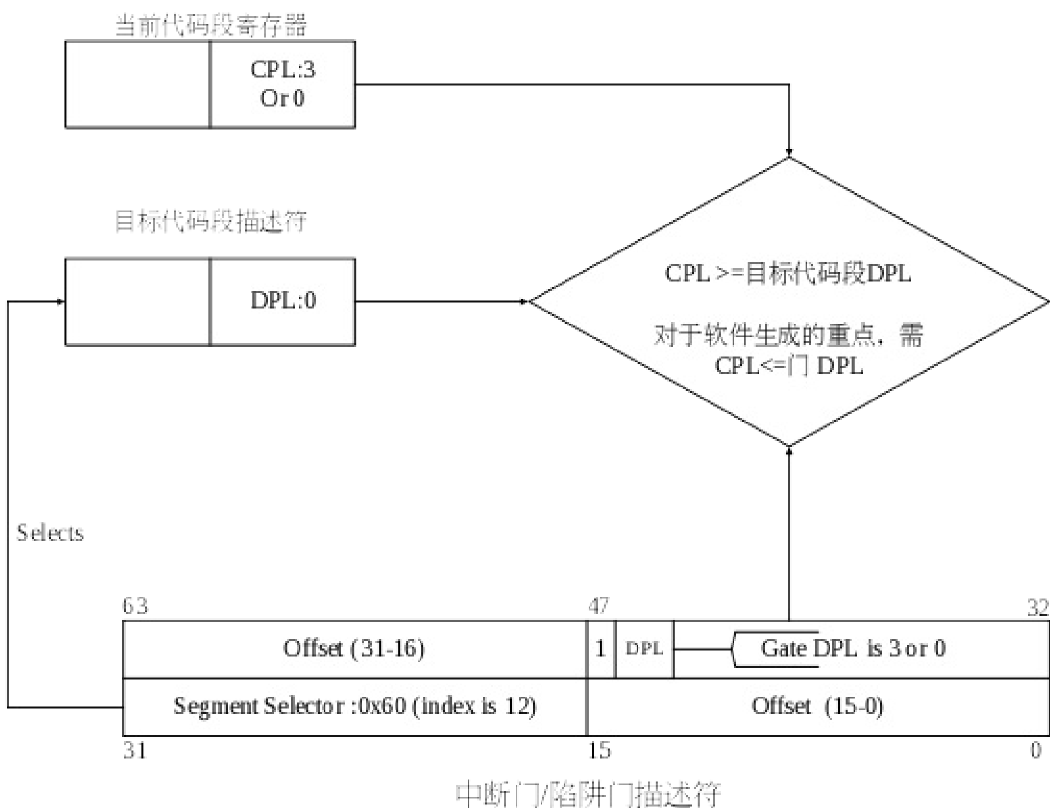


图 中断发生时实施特权检查的过程

门中的DPL和段选择符一起控制着访问，同时，段选择符结合偏移量（Offset）指出了中断处理例程的入口点。内核一般在门描述符中填入内核代码段的段选择子。产生中断后，CPU一定不会将运行控制从高特权级转向低特权级，特权级必须要么保持不变（当操作系统内核自己被中断的时候），或被提升（当用户态程序被中断的时候）。无论哪一种情况，作为结果的CPL必须等于目的代码段的DPL。如果CPL发生了改变（比如从用户态到内核态），一个栈切换操作（通过TSS完成）就会发生。如果中断是被用户态程序中的指令所触发的（比如软件执行INT n生产的中断），还会增加一个额外的检查：门的DPL必须具有与CPL相同或更低的特权。这就防止了用户代码随意触发中断。如果这些检查失败，就会产生一个一般保护异常（general-protection exception）。

【实现】初始化中断控制器

80386把中断号0～31分配给陷阱、故障和非屏蔽中断，而把32～47之间的中断号分配给可屏蔽中断。可屏蔽中断的中断号是通过对中断控制器的编程来设置的。下面描述了对8259A中断控制器初始化过程。

8259A通过两个I/O地址来进行中断相关的数据传送，对于单个的8259A或者是两级级联中的主8259A而言，这两个I/O地址是0x20和0x21。对于两级级联的从8259A而言，这两个I/O地址是0xA0和0xA1。8259A有两种编程方式，一是初始化方式，二是工作方式。在操作系统启动时，需要对8959A做一些初始化工作，即实现8259A的初始化方式编程。8259A中的四个中断命令字（ICW）寄存器用来完成初始化编程，其含义如下：

- ICW1：初始化命令字。
- ICW2：中断向量寄存器，初始化时写入高五位作为中断向量的高五位，然后在中断响应时由8259根据中断源（哪个管脚）自动填入形成完整的8位中断向量（或叫中断类型号）。
- ICW3： 8259的级联命令字，用来区分主片和从片。
- ICW4：指定中断嵌套方式、数据缓冲选择、中断结束方式和CPU类型。

8259A初始化的过程就是写入相关的命令字，8259A内部储存这些命令字，以控制8259A工作。有关的硬件可看附录补充资料。这里只把ucore对8259A的初始化过程（在picirq.c中的pic_init函数实现）描述一下:

```
//此时系统尚未初始化完毕，故屏蔽主从8259A的所有中断

outb(IO_PIC1 + 1, 0xFF);
outb(IO_PIC2 + 1, 0xFF);

// 设置主8259A的ICW1，给ICW1写入0x11，0x11表示（1）外部中断请求信号为上升沿触发有效，（2）系统中有多片8295A级联，（3）还表示要向ICW4送数据

// ICW1设置格式为:  0001g0hi
//    g:  0 = edge triggering, 1 = level triggering
//    h:  0 = cascaded PICs, 1 = master only
//    i:  0 = no ICW4, 1 = ICW4 required

outb(IO_PIC1, 0x11);

// 设置主8259A的ICW2:  给ICW2写入0x20，设置中断向量偏移值为0x20，即把主8259A的IRQ0-7映射到向量0x20-0x27

outb(IO_PIC1 + 1, IRQ_OFFSET);

// 设置主8259A的ICW3:  ICW3是8259A的级联命令字，给ICW3写入0x4，0x4表示此主中断控制器的第2个IR线（从0开始计数）连接从中断控制器。
```

```
outb(IO_PIC1 + 1, 1 << IRQ_SLAVE);

//设置主8259A的ICW4：给ICW4写入0x3，0x3表示采用自动EOI方式，即在中断响应时，在8259A送出中断矢量后，自动将ISR相应位复位；并且采用一般嵌套方式，即当某个中断正在服务时，本级中断及更低级的中断都被屏蔽，只有更高的中断才能响应。

// ICW4设置格式为:  000nbmap
//    n:  1 = special fully nested mode
//    b:  1 = buffered mode
//    m:  0 = slave PIC, 1 = master PIC
//      (ignored when b is 0, as the master/slave role
//      can be hardwired).
//    a:  1 = Automatic EOI mode
//    p:  0 = MCS-80/85 mode, 1 = intel x86 mode
outb(IO_PIC1 + 1, 0x3);

//设置从8259A的ICW1：含义同上

outb(IO_PIC2, 0x11);    // ICW1

//设置从8259A的ICW2：给ICW2写入0x28，设置从8259A的中断向量偏移值为0x28

outb(IO_PIC2 + 1, IRQ_OFFSET + 8);    // ICW2

//0x2表示此从中断控制器链接主中断控制器的第2个IR线

outb(IO_PIC2 + 1, IRQ_SLAVE);    // ICW3

//设置主8259A的ICW4：含义同上

outb(IO_PIC2 + 1, 0x3);    // ICW4

//设置主从8259A的OCW3：即设置特定屏蔽位（值和英文解释不一致），允许中断嵌套；不查询；将读入其中断请求寄存器IRR的内容

// OCW3设置格式为:  0ef01prs
//   ef:  0x = NOP, 10 = clear specific mask, 11 = set specific mask
//    p:  0 = no polling, 1 = polling mode
//   rs:  0x = NOP, 10 = read IRR, 11 = read ISR
outb(IO_PIC1, 0x68);    // clear specific mask
outb(IO_PIC1, 0x0a);    // read IRR by default

outb(IO_PIC2, 0x68);    // OCW3
outb(IO_PIC2, 0x0a);    // OCW3

//初始化完毕，使能主从8259A的所有中断

if (irq_mask != 0xFFFF) {
    pic_setmask(irq_mask);
}
```

【实现】初始化中断门描述符表

ucore操作系统如果要正确处理各种不同的中断事件，就需要安排应该由哪个中断服务例程负责处理特定的中断事件。系统将所有的中断事件统一进行了编号（0～255），这个编号称为中断号或中断向量。

为了完成中断号和中断服务例程起始地址的对应关系，首先需要建立256个中断处理例程的入口地址。为此，通过一个 C程序 tools/vector.c 生成了一个文件vectors.S，在此文件中的__vectors地址处开始处连续存储了256个中断处理例程的入口地址数组，且在此文件中的每个中断处理例程的入口地址处，实现了中断处理过程的第一步初步处理。

有了中断服务例程的起始地址，就可以建立对应关系了，这部分的实现在trap.c文件中的idt_init函数中实现：

```
//全局变量：中断门描述符表

static struct gatedesc idt[256] = {{0}};
……
void idt_init(void) {

//保存在vectors.S中的256个中断处理例程的入口地址数组

    extern uint32_t __vectors[];
    int i;

//在中断门描述符表中通过建立中断门描述符，其中存储了中断处理例程的代码段GD_KTEXT和偏移量\__vectors[i]，特权级为DPL_KERNEL。这样通过查询idt[i]就可定位到中断服务例程的起始地址。

    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 0, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }

//建立好中断门描述符表后，通过指令lidt把中断门描述符表的起始地址装入IDTR寄存器中，从而完成中段描述符表的初始化工作。

    lidt(&idt_pd);
}
```

【实现】外设的相关中断初始化

串口的初始化函数serial_init（位于/kern/driver/console.c）中涉及中断初始化工作的很简单：

```
......
// 使能串口1接收字符后产生中断
    outb(COM1 + COM_IER, COM_IER_RDI);
......
// 通过中断控制器使能串口1中断
pic_enable(IRQ_COM1);
```

键盘的初始化函数kbd_init（位于kern/driver/console.c中）完成了对键盘的中断初始化工作，具体操作更加简单：

```
......
// 通过中断控制器使能键盘输入中断
pic_enable(IRQ_KBD);
```

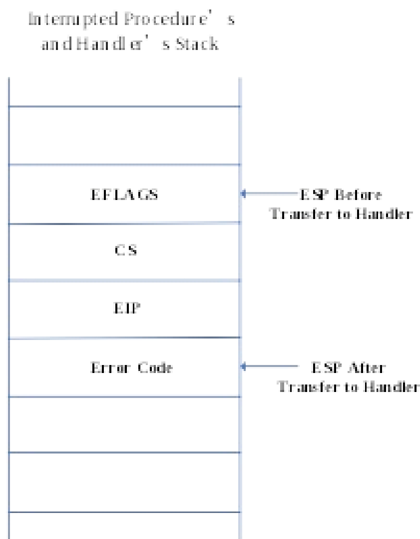
时钟是一种有着特殊作用的外设，其作用并不仅仅是计时。在后续章节中将讲到，正是由于有了规律的时钟中断，才使得无论当前CPU运行在哪里，操作系统都可以在预先确定的时间点上获得CPU控制权。这样当一个应用程序运行了一定时间后，操作系统会通过时钟中断获得CPU控制权，并可把CPU资源让给更需要CPU的其他应用程序。时钟的初始化函数clock_init（位于kern/driver/clock.c中）完成了对时钟控制器8253的初始化：

```
    ......
//设置时钟每秒中断100次
    outb(IO_TIMER1, TIMER_DIV(100) % 256);
    outb(IO_TIMER1, TIMER_DIV(100) / 256);
// 通过中断控制器使能时钟中断
    pic_enable(IRQ_TIMER);
```

【实现】中断处理过程

当中断产生后，首先硬件要完成一系列的工作（如小节“中断处理中硬件负责完成的工作”所描述的“硬件中断处理过程1（起始）”内容），由于中断发生在内核态执行过程中，所以特权级没有变化，所以CPU在跳转到中断处理例程之前，还会在内核栈中依次压入错误码（可选）、EIP、CS和EFLAGS，下图显示了在相同特权级下中断产生后的栈变化示意图：

统一特权级下中断产生后的栈变化



然后CPU就跳转到IDT中记录的中断号i所对应的中断服务例程入口地址处继续执行。 vector.S文件中定义了每个中断的中断处理例程的入口地址 (保存在 vectors 数组中)。其中，中断可以分成两类：一类是压入错误编码的 (error code)，另一类不压入错误编码。对于第二类， vector.S 自动压入一个 0。此外，还会压入相应中断的中断号。在内核栈中压入一个或两个必要的参数之后，都会跳转到统一的入口 __alltraps 处（位于trapentry.S中）继续执行。

CPU从__alltraps处开始，在栈中按照trapframe结构压入各个寄存器，此时内核栈的结构如下所示：

```
uint32_t reg_edi;
uint32_t reg_esi;
uint32_t reg_ebp;
uint32_t reg_oesp;            /* Useless */
uint32_t reg_ebx;
uint32_t reg_edx;
uint32_t reg_ecx;
uint32_t reg_eax;
uint16_t tf_es;
uint16_t tf_padding1;
uint16_t tf_ds;
uint16_t tf_padding2;
uint32_t tf_trapno;
/* below here defined by x86 hardware */
uint32_t tf_err;
uintptr_t tf_eip;
uint16_t tf_cs;
uint16_t tf_padding3;
uint32_t tf_eflags;
```

此时，为了将来能够恢复被打断的内核执行过程所需的寄存器内容都保存好了。为了正确进行中断处理，把DS和ES寄存器设置为GD_KDATA，这是为了预防从用户态产生的中断（当然，到目前为止，ucore都在内核态执行，还不会发生这种情况）。把刚才保存的trapframe结构的起始地址（即当前SP值）压栈，然后调用 trap函数（定义在trap.c中），就开始了对具体中断的处理。trap进一步调用trap_dispatch函数，完成对具体中断的处理。在相应的处理过程结束以后，trap将会返回，在__trapret:中，完成对返回前的寄存器和栈的回复准备工作，最后通过iret指令返回到中断打断的地方继续执行。整个中断处理流程大致如下：

<table>
<tr><th>trapasm.S</th><th>trap.c</th></tr>
<tr><td>
1)产生中断后，CPU 跳转到相应的中断处理入口(vectors)，并在栈中压入相应的 error_code（是否存在与异常号相关）以及 trap_no，然后跳转到 alltraps 函数入口：
注意：此处的跳转是 jmp 过程
<table>
<tr><td>(high)</td><td>...</td></tr>
<tr><td>产生中断时的 eip →</td><td>eip</td></tr>
<tr><td></td><td>error_code</td></tr>
<tr><td>esp →</td><td>trap_no</td></tr>
<tr><td>(low)</td><td>.
.</td></tr>
</table>
在栈中保存当前被打断程序的 trapframe 结构(参见过程 trapasm.S)。设置 kernel (内核) 的数据段寄存器，最后压入 esp，作为 trap 函数参数(struct trapframe * tf) 并跳转到中断处理函数 trap 处：
注意：此时的跳转是 call 调用，会压入返回地址 eip，注意区分此处 eip 与 trapframe 中 eip：
trapframe 的结构为：
<table>
<tr><td>Struct trapframe
{
uint edi;
uint esi;
uint ebp;
…
ushort es;
ushort padding1;
ushort ds;
ushort padding2;
uint trapno;
uint err;
uint eip;
...
}</td><td>观察 trapframe 结构与中断产生过程的压栈顺序。
需要明确 pushal 指令都保存了哪些寄存器，按照什么顺序？

← trap_no
← trap_error
← 产生中断处的 eip</td></tr>
</table>
进入 trap 函数，对中断进行相应的处理：
</td><td></td></tr>
<tr><td></td><td>
2)详细的中断分类以及处理流程如下：
根据中断号对不同的中断进行处理。其中，若中断号是 IRQ_OFFSET + IRQ_TIMER 为时钟中断，则把 ticks 将增加一。
若中断号是 IRQ_OFFSET + IRQ_COM1 为串口中断，则显示收到的字符。
若中断号是 IRQ_OFFSET + IRQ_KBD 为键盘中断，则显示收到的字符。
若为其他中断且产生在内核状态，则挂起系统；
</td></tr>
<tr><td>
3)结束 trap 函数的执行后，通过 ret 指令返回到 alltraps 执行过程。
从栈中恢复所有寄存器的值。
调整 esp 的值：跳过栈中的 trap_no 与 error_code，使 esp 指向中断返回 eip，通过 iret 调用恢复 cs、eflag 以及 eip，继续执行。
</td><td></td></tr>
</table>

可在内核态和用户态之间进行切换的ucore

在操作系统原理中，一直强调操作系统运行在内核态（特权态），应用程序运行在用户态（非特权态）。但为什么说处于用户态的应用程序就不能访问内核态的数据，而内核态的操作系统可以访问用户态的数据？我们有没有一个project来体验内核态和用户态的区别是什么？更进一步体验如何在内核态和用户态之间进行切换呢？project4.1.1为此进行了尝试。

实验目标

通过学习和实践，读者可以了解如何CPU处不同特权态下的执行特点和限制，理解如何从内核态切换到用户态，以及如何从用户态切换到内核态。

proj4.1.1概述

实现描述

proj4.1.1建立在proj4.1（当然是基于proj4）基础之上，主要完成了用户态（非特权态）与内核态相互切换的过程。相对于proj4，主要增加了两部分工作，一部分是从用户态返回内核态的准备工作，即建立任务段（Task Segment）和任务段描述符（SEG_TSS），设置陷阱中断号（T_SWITCH_TOK）和对应的中断处理例程。另外一部分是对内核栈进行各种特殊处理，使得能够完成内核态切换到用户态或用户态切换到内核态的工作。

项目组成

这里我们通过proj4.1.1来完成此事。proj4.1.1整体目录结构如下所示：

```
proj4.1.1
|-- kern
|   |-- init
|   |   `-- init.c
|   |-- mm
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   `-- pmm.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
……
```

相对于proj4，改动不多，主要修改和增加的文件如下：

- memlayout.h：定义了全局描述符的索引值和一些段描述符的属性。
- pmm.[ch]：为了能够使CPU从用户态转换到内核态，在ucore初始化时，设置任务段和任务段描述符，重新加载任务段和段描述符表；
- trap.c：设置自定义的陷阱中断T_SWITCH_TOK（用于用户态切换到内核态）和实现对自定义的陷阱中断T_SWITCH_TOK/T_SWITCH_TOU的中断处理例程，使得CPU能够在内核态和用户态之间切换。

编译运行

编译并运行proj4.1.1的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
QEMU
Special kernel symbols:
  entry  0x00100000 (phys)
  etext  0x001031fe (phys)
  edata  0x0010e998 (phys)
  end    0x0010fca0 (phys)
Kernel executable memory footprint: 64KB
++ setup timer interrupts
@ring 0
 cs = 8
 ds = 10
 es = 10
 ss = 10
--- switch to  user  mode ---
@ring 3
 cs = 1b
 ds = 23
 es = 23
 ss = 23
--- switch to kernel mode ---
@ring 0
 cs = 8
 ds = 10
 es = 10
 ss = 10
```

通过上图，我们可以看到ucore在切换到用户态之前，先显示了当前CPU的特权级（CS的最低两位），CS/DS/ES/SS的值（即对应的段描述符表的索引值），可以看到特权级为0。根据lgdt函数（位于kern/mm/pmm.c中）的处理，CS的值是内核代码段描述符的索引下标，DS/ES/SS的值是内核数据段描述符的索引下标；而在切换到用户态后，又显示了一下，当前CPU的特权级为3， CS的值为1b，DS/ES/SS的值为23，把这四个寄存器的值&0xfc，则分别为0x18（SEG_UTEXT）和0x20（SEG_UDATA），说明确实运行在用户态了。在执行了系统调用T_SWITCH_TOK后，又回到了内核态。下面我们将分析到底发生了什么事情。

【背景】分段机制的特权限制

在保护模式下，特权级总共有4个，编号从0（最高特权）到3（最低特权）。三类主要的资源，即内存、I/O地址空间以及特权指令需要保护。特权指令如果被用户态的程序所使用，就会受到保护模式的保护机制限制，导致一个故障中断（general-protection exception）。对内存和I/O端口的访问存在类似的特权级限制。为了更好地理解不同特权级，这里先介绍三个概念

- CPL：当前特权级（Current Privilege Level) 保存在CS段寄存器（选择子）的最低两位，CPL就是当前活动代码段的特权级，并且它定义了当前所执行程序的特权级别）
- DPL：描述符特权（Descriptor Privilege Level） 存储在段描述符中的权限位，用于描述对应段所属的特权等级，也就是段本身真正的特权级。
- RPL：请求特权级RPL(Request Privilege Level) RPL保存在选择子的最低两位。RPL说明的是进程对段访问的请求权限，意思是当前进程想要的请求权限。RPL的值由程序员自己来自由的设置，并不一定RPL>=CPL，但是当RPL\<CPL时，实际起作用的就是CPL了，因为访问时的特权检查是判断：max(RPL,CPL)<=DPL是否成立，所以RPL可以看成是每次访问时的附加限制，RPL=0时附加限制最小，RPL=3时附加限制最大。

【背景】80386的任务切换

`任务`是80386硬件描述中的一个名词，在这里我们可以简单地把运行在内核态的ucore成为一个任务，把运行在用户态的应用称为另外一任务。任务寄存器(Task Register，简称TR) 储存了一个16位的选择子(对软件可见)，用来索引全局描述符表(GDT)中的一项。TR对应的描述符描述的一个任务状态段(TSS:Task Status Segment)。

TSS 任务状态段(Task State Segment，简称TSS)。任务状态段（TSS）是位于GDT中的一个系统段描述符。任务状态段是做什么的呢？任务状态段就是内存中的一个数据结构。这个结构中保存着和任务相关的信息。当发生任务切换的时候会把当前任务用到的寄存器内容(CS/EIP/ DS/SS/EFLAGS...)等保存在TSS中以便任务切换回来时候继续使用。ucore根据80386硬件手册建立的TSS数据结构如下所示：

```
struct taskstate {

    uint32_t ts_link;        // 链接字段
    uintptr_t ts_esp0;        // 0级栈指针
    uint16_t ts_ss0;        // 0级栈段寄存器
    uint16_t ts_padding1;
    uintptr_t ts_esp1;
    uint16_t ts_ss1;
    uint16_t ts_padding2;
    uintptr_t ts_esp2;
    uint16_t ts_ss2;
    uint16_t ts_padding3;
    physaddr_t ts_cr3;        // 页目录基址寄存器
    uintptr_t ts_eip;        // 切换的上次EIP
    uint32_t ts_eflags;
    uint32_t ts_eax;        // 保存的通用寄存器eax
    uint32_t ts_ecx;
    uint32_t ts_edx;
    uint32_t ts_ebx;
    uintptr_t ts_esp;
    uintptr_t ts_ebp;
    uint32_t ts_esi;
    uint32_t ts_edi;
    uint16_t ts_es;            // 保存的段寄存器
    uint16_t ts_padding4;
    uint16_t ts_cs;
    uint16_t ts_padding5;
    uint16_t ts_ss;
    uint16_t ts_padding6;
    uint16_t ts_ds;
    uint16_t ts_padding7;
    uint16_t ts_fs;
    uint16_t ts_padding8;
    uint16_t ts_gs;
    uint16_t ts_padding9;
    uint16_t ts_ldt;
    uint16_t ts_padding10;
    uint16_t ts_t;            // 调试陷阱标志(只用位0)
    uint16_t ts_iomb;        // i/o map 基地址
};
```

从上图中可以 ，TSS的基本格式由104字节组成。这104字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。基本的104字节可分为链接字段区域、内层栈指针区域、地址映射寄存器区域、寄存器保存区域和其它字段等五个区域。

其中比较重要的是内层栈指针区域，为了有效地实现保护，同一个任务在不同的特权级下使用不同的栈。例如，当从外层特权级3变换到内层特权级0时，任务使用的栈也同时从3级变换到0级栈；当从内层特权级0变换到外层特权级3时，任务使用的栈也同时从0级栈变换到3级栈。所以ucore使用的是0级栈，用户态应用使用的是3级栈。 TSS的内层栈指针区域中有三个

栈指针，它们都是48位的全指针(16位的选择子和32位的偏移)，分别指向0级、1级和2级栈的栈顶，依次存放在TSS中偏移为4、12及20开始的位置。当发生从3级向0级转移时，把0级栈指针装入0级的SS及ESP寄存器以变换到0级栈。没有指向3级栈的指针，因为3级是最外层，所以任何一个向内层的转移都不可能转移到3级。但是，当特权级由0级向3级变换时，并不把0级栈的指针保存到TSS的栈指针区域。这表明向3级向0级转移时，总是把0级栈认为是一个空栈。

当发生任务切换时，80386中各寄存器的当前值被自动保存到TR所指定的TSS中，然后下一任务的TSS的选择子被装入TR；最后，从TR所指定的TSS中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在TSS中保存任务现场各寄存器状态的完整映象，实现任务的切换。

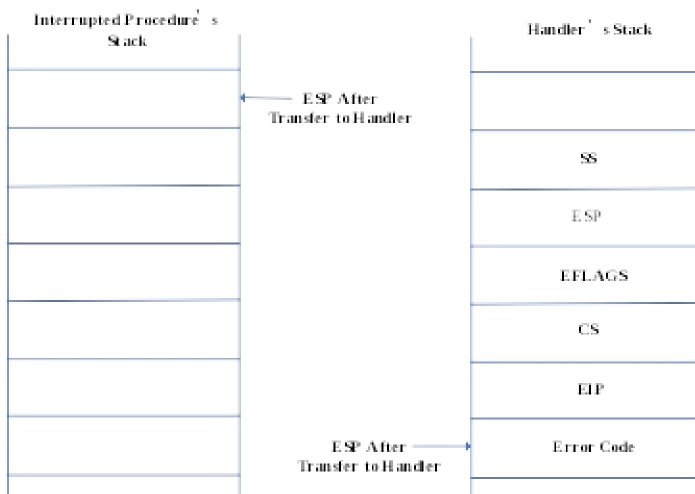
【实现】内核态切换到用户态

在kern/init.c中的switch_test函数完成了内核态<-->用户态之间的切换。内核态切换到用户态是通过swtich_to_user函数，执行指令“int T_SWITCH_TOU”。当CPU执行这个指令时，由于是在switch_to_user执行在内核态，所以不存在特权级切换问题，硬件只会在内核栈中压入Error Code（可选）、EIP、CS和EFLAGS（如下图所示），然后跳转到到IDT中记录的中断号T_SWITCH_TOU所对应的中断服务例程入口地址处继续执行。通过2.3.7小节“中断处理过程”可知，会执行到trap_disptach函数（位于trap.c）：

```
case T_SWITCH_TOU:
if (tf->tf_cs != USER_CS) {
//当前在内核态，需要建立切换到用户态所需的trapframe结构的数据switchk2u
switchk2u = *tf;
switchk2u.tf_cs = USER_CS;
switchk2u.tf_ds = switchk2u.tf_es = switchk2u.tf_ss = USER_DS;
switchk2u.tf_esp = (uint32_t)tf + sizeof(struct trapframe) - 8;
   //设置EFLAG的I/O特权位，使得在用户态可使用in/out指令
switchk2u.tf_eflags |= (3 << 12);
    //设置临时栈，指向switchk2u，这样iret返回时，CPU会从switchk2u恢复数据，
//而不是从现有栈恢复数据。
    *((uint32_t *)tf - 1) = (uint32_t)&switchk2u;
}
```

这样在trap将会返回，在__trapret:中，根据switchk2u的内容完成对返回前的寄存器和栈的回复准备工作，最后通过iret指令，CPU返回“int T_SWITCH_TOU”的后一条指令处，以用户态模式继续执行。

有特权级变化的中断产生后堆栈内容



【实现】用户态切换到内核态

CPU在用户态执行到switch_to_kernel()函数（即执行“int T_SWITCH_TOK”）时，由于当前处于用户态，而中断产生后，CPU会进入内核态，所以存在特权级转换。硬件会在内核栈中压入Error Code（可选）、EIP、CS和EFLAGS、ESP（用户态特权级的）、SS（用户态特权级的）（如下图所示），然后跳转到到IDT中记录的中断号T_SWITCH_TOK所对应的中断服务例程入口地址处继续执行。通过2.3.7小节“中断处理过程”可知，会执行到trap_disptach函数（位于trap.c）：

```
case T_SWITCH_TOK:
    if (tf->tf_cs != KERNEL_CS) {
//发出中断时，CPU处于用户态，我们希望处理完此中断后，CPU继续在内核态运行，
//所以把tf->tf_cs和tf->tf_ds都设置为内核代码段和内核数据段
      tf->tf_cs = KERNEL_CS;
      tf->tf_ds = tf->tf_es = KERNEL_DS;
      //设置EFLAGS，让用户态不能执行in/out指令
      tf->tf_eflags &= ~(3 << 12);

      switchu2k = (struct trapframe *)(tf->tf_esp - (sizeof(struct trapframe) - 8));
      //设置临时栈，指向switchu2k，这样iret返回时，CPU会从switchu2k恢复数据，
//而不是从现有栈恢复数据。
      memmove(switchu2k, tf, sizeof(struct trapframe) - 8);
      *((uint32_t *)tf - 1) = (uint32_t)switchu2k;
     }
     break;
```

这样在trap将会返回，在__trapret:中，根据switchk2u的内容完成对返回前的寄存器和栈的回复准备工作，最后通过iret指令，CPU返回“int T_SWITCH_TOU”的后一条指令处，以内核态模式继续执行。

原理归纳与小结

读者通过阅读本章和做实验，相信已经尝试了加载操作系统、操作系统访问外设、操作系统处理中断、用户态与内核态切换等一系列小project。这里面体现了哪些操作系统的原理和概念呢？

系统软件：在这里读者应该体会到操作系统是一个软件，采用编译器生成可执行代码，通过bootloader加载到内存中执行，可以执行所有CPU指令（包括特权指令），访问到所有硬件资源。相对于一般的应用软件，它主要完成的工作是对计算机的硬件资源管理，并给上层应用提供服务（这里还体现不够）。所以我们把操作系统看成是提供计算机系统级管理和基础共性服务的一种系统软件。

段式的内存管理：在这里读者应该体会到CPU访问的地址首先是一个虚拟地址，然后通过MMU的段式地址保护检查和变换（这要看段描述符表中的描述符是如何设置段基址和段范围的）才能把虚拟地址变换成线性地址（由于没有启动分页机制，线性地址就是物理地址），如果虚拟地址的偏移值超过了段范围值，这会出现故障中断。

中断处理：在这里读者应该体会到中断包含了外设产生的外设中断（来的时机不确定）和CPU主动产生的陷阱中断（执行特定指令就会产生），有了中断，操作系统就可以随时打断CPU正常的顺序执行，转而处理相对更加紧急的中断事件。为了能够让CPU继续正常执行，在中断处理前需要保存足够的硬件信息（主要是CPU各种寄存器的值，一般放在内核栈中），以便于中断处理完毕后，通过恢复这些硬件信息，能够回到被打断的地方继续执行。

特权级：在这里读者应该体会到在不同的特权级可以完成的事情是不一样的。操作系统需要管理整个计算机资源，所以它应该运行在CPU的最高特权级，而应用程序不必也不能使用特权指令或访问操作系统的地址空间，所以它应该运行在CPU的非特权级。如果应用程序执行特权指令或访问操作系统的的地址空间，则CPU硬件的安全检查机制会阻止应用程序的执行并尝试故障中断，把CPU控制权转给操作系统进行进一步处理。如果应用程序需要访问计算机资源，可以通过执行特定的指令，产生陷阱中断，从而完成从用户态到内核态的切换，让操作系统为应用程序提供服务。

物理内存管理

一个编程人员希望拥有容量无限大、速度无限快，而且是非易失型的（nonvolatile）的内存空间，到lab1为止，这个梦想还无法轻易满足。为此绝大多数的计算机采用了一种折衷的方法，即建立一个分层的存储器结构，最高层是CPU内部的一些寄存器，它们的访问速度是最快的，但容量不是很大，一般小于1KB；第二层是高速缓存（即硬件cache），实现在CPU内部，接近寄存器速度，容量一般小于4MB；第三层是主存储器（内存），其访问速度比寄存器小一个数量级、价格便宜，目前几百块人民币就可以买到4GB。以上这三种存储器都是易失型的，即在断电后，其内容全部会丢失掉。第四层是磁盘，它的访问速度较慢、价格较便宜，目前花几百块钱就可以买到存储容量>1TB的硬盘，而且是非易失型的。

操作系统需要尽量满足编程人员的梦想，为此它需要管理上述存储器层次结构形成的存储空间，并完成如下主要任务：

- 记录存储空间的使用情况，即记录哪些部分正在被使用，哪些部分还空闲；
- 当需求方需要存储空间时，能快速地分配给它合适大小的空间；在需求方显式表示不需要申请到的存储空间时，能把存储空间回收，便于以后的分配；
- 隔离不同的内存区域，确保在限制在一个内存区域中运行的软件无法访问区域以外的内存空间。这种机制称为地址保护（地址隔离）机制。
- 如果内存太小，就需要把内存当中使用较少的数据所占空间送到磁盘上，给使用较多的数据腾出内存空间来；如果将来又访问到缓存到硬盘的数据，需要把这些数据重新加载到内存中进行访问。这种机制称为换入换出（swap in/out）机制，并涉及页替换算法。
- 即使需求方表明了需要内存，但如果需求方没有实际访问所需内存前，则并不完成实际的物理内存分配。这种机制称为按需分配（如果是基于很分页机制，也称为按需分页）。
- 设两个具有父子关系的程序共享同一地址空间（子程序同享父程序的地址空间），若二程序只是读此地址空间，则地址空间不会有变化；若其中一个程序对此地址空间某地址进行了写操作，则要把包含此地址的页空间复制一份给执行写操作的程序，这时此二程序将有不同的地址空间，可独立运行，相互不干扰。这种机制减少了父程序创建子程序地址空间的开销，称为写时复制（Copy On Write，简称COW）机制。

本章内容主要涉及操作系统的内存管理，包括物理内存管理和基于分页机制的虚拟内存管理。读者通过阅读本章的内容并动手实践相关的5个project实验：

- proj5：能够探测物理内存并建立页表，实现分页管理
- proj5.1/5.1.1/5.1.2：实现基于连续物理页的first/best/worst-fit分配算法
- proj5.2：实现基于连续物理页的buddy分配算法
- proj6：实现任意大小内存分配的slab分配算法
- proj7：实现缺页中断服务例程和虚拟内存管理结构（VMM struct），提供按需分页的支持

- proj8：实现类似改进时钟算法的页面置换算法并支持页粒度的换入换出机制
- proj9/proj9.1/proj9.2：完善虚存管理(proj9)并逐步实现了进程间内存共享（proj9.1）和copy on write（COW）机制（proj9.2）

可以掌握如下知识：

- 与操作系统原理相关
- 内存管理：基于分页机制的内存管理
- 内存管理：连续内存分配算法
- 内存管理：非连续内存分配算法
- 内存管理和中断：缺页中断服务例程
- 内存管理：虚存管理中的页面置换算法和页换入换出机制
- 内存管理：按需分页机制和写时复制机制
- 操作系统原理之外
- 80386对分页管理（页表等）的硬件支持
- 页粒度的页面置换策略和页换入换出的具体实现

本章内容中主要涉及内存管理的重要功能主要有两个：

- 提供空闲内存空间：这样给操作系统和应用程序的代码和数据足够的存放“地方”，使得二者能够正常高效地运行，为此需要完成内存/外存的空间的分配、管理、释放等算法。
- 提供内存空间隔离保护：隔离用户态应用程序和内核态操作系统之间，以及不同应用程序之间的内存空间，使得不会出现访问冲突，为此需要为不同的应用程序和操作系统划分不同的地址空间，一个应用程序越界规定的地址空间会出现内存访问故障中断。

为了让读者能够从实践上来理解内存管理的基本原理，我们设计了上述实验，主要的功能逐步实现如下所示：

- 首先是扩展ucore的功能，使它能够发现并管理PC系统中可用的空闲物理内存；
- 然后是建立分页机制，建立线性地址（分段机制已经完成了逻辑地址到线性地址的转换）到物理地址的映射关系和具体转换操作，这样使得应用程序无法直接访问到物理地址，而是只能访问由操作系统设定好的物理地址空间，从而使得应用程序的访问空间可控；
- 为了高效地完成操作系统的其他功能单元和应用程序的空闲内存空间需求，需要设计以页（4096字节）为最小分配单位的面向连续物理地址空间的内存分配算法； 还要设计面向任意大小的内存空间（在物理地址空间上不一定连续）的虚拟内存分配算法；
- 为了给应用程序提供超过实际物理内存空间大小的虚拟内存空间，需要把临时不常用到的内存换出（swap out）到硬盘（也称外存）中，等到需要访问的时候，再换入（swap in）到内存中。设计高效的页面置换算法会尽量保存经常访问的数据在内存中，而不经常访问的数据会换出到硬盘中。

试验目标

操作系统和应用程序都需要内存空间来存放代码和数据，这要求操作系统能够高效管理和保护整个计算机中的物理内存，并给自己和上层应用提供简洁安全的内存申请和释放的服务接口。通过分段机制可以完成虚拟地址到线性地址的转换，而通过分页机制可以进一步完成线性地址到物理地址的转换。分段机制中对每段的大小是可变的，分页机制中页的大小固定为4KB，这样在操作系统对实现内存管理上会更加简洁。所以在建立好分页机制后，分段机制的映射功能退化为对等映射，即虚拟地址=线性地址，这样实际的地址映射工作将由分页机制来完成。

为此ucore需要在已有的分段机制的基础上，进一步加入分页机制，达到可以通过分页完成对不同应用程序执行的内存空间进行隔离（其实分段也能够达到此目的，但相对实现的开销会比较大，受到的限制（比如支持的应用执行个数）也较多）的目标。并且为后续的虚存管理提供基础支持。

proj5/5.1/5.1.1/5.1.2/5.2概述

实现描述

proj5基于proj4.3实现，主要完成了对计算机实际物理内存大小与分布的探测，实现以页（大小为4KB）为单位的简单物理内存管理，并通过建立二级页表，实现了分页内存管理，为将来试验中实现虚存管理打下一个基础。通过分析和实现proj5，读者可以了解到：

- 物理内存空间布局的探测；
- 基于分页机制的存储管理；
- 32位地址空间的二级页表结构；

proj5.1在proj5的基础上实现了基于best fit内存分配算法的页级内存分配和释放功能；proj5.1.1在proj5.1的基础上实现了first fit内存分配算法的页级内存分配和释放功能；proj5.1.2在proj5.1的基础上实现了worst fit内存分配算法的页级内存分配和释放功能；proj5.2在proj5.1的基础上实现了更加实用和强大的buddy内存分配算法的页级内存分配和释放功能。这些proj是操作系统原理相关算法的具体实现。读者可以参考这些实现完成新的内存分配算法。在这里通过讲解proj5的实现，让读者理解如何基于一个内存分配管理框架来实现不同的内存分配算法。

项目组成

```
proj5
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- kern
|   |-- init
|   |   |-- entry.S
|   |   `-- init.c
|   |-- mm
|   |   |-- default_pmm.c
|   |   |-- default_pmm.h
|   |   |-- memlayout.h
|   |   |-- mmu.h
|   |   |-- pmm.c
|   |   `-- pmm.h
|   |-- sync
|   |   `-- sync.h
|   `-- trap
|       |-- trap.c
|       |-- trapentry.S
|       |-- trap.h
|       `-- vectors.S
|-- libs
|   |-- atomic.h
|   |-- list.h
`-- tools
    |-- kernel.ld
```

相对与proj4.3，proj5增加了6个文件。主要修改和增加的文件如下：

- boot/bootasm.S：增加了对计算机系统中物理内存布局的探测功能；
- kern/init/entry.S：根据临时段表重新暂时建立好新的段空间，为进行分页做好准备。
- kern/mm/default_pmm.[ch]：提供基本的基于链表方法的物理内存管理（分配单位为页，即4096字节）；
- kern/mm/pmm.[ch]：pmm.h定义物理内存管理类框架struct pmm_manager，基于此通用框架可以实现不同的物理内存管理策略和算法(default_pmm.[ch] 实现了一个基于此框架的简单物理内存管理策略)； pmm.c包含了对此物理内存管理类框架的访问，以及与建立、修改、访问页表相关的各种函数实现。
- kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存eflag寄存器中的中断屏蔽位信息并屏蔽中断的功能，另一个是根据保存的中断屏蔽位信息来使能中断的功能；
- libs/list.h：定义了通用双向链表结构以及相关的查找、插入等基本操作，这是建立基于链表方法的物理内存管理（以及其他内核功能）的基础。其他有类似双向链表需求的内核功能模块可直接使用list.h中定义的函数。
- libs/atomic.h：定义了对一个变量进行读写的原子操作，确保相关操作不被中断打断。

- tools/kernel.ld：修改了ucore的起始入口和代码段的起始地址

编译运行

编译并运行proj5的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
chenyu@chenyu-laptop:~/oscourse/ucore-svn/lab2_memory/proj5$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc010537f (phys)
  edata  0xc01169b8 (phys)
  end    0xc01178dc (phys)
Kernel executable memory footprint: 95KB
memory managment: default_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
++ setup timer interrupts
100 ticks
100 ticks
……
```

通过上图，我们可以看到ucore在显示其entry（入口地址）、etext（代码段截止处地址）、edata（数据段截止处地址）、和end（ucore截止处地址）的值后，探测出计算机系统中的物理内存的布局（e820map下的显示内容）。接下来ucore会以页为最小分配单位实现一个简单

的内存分配管理，完成二级页表的建立，进入分页模式，执行各种我们设置的检查，最后显示ucore建立好的二级页表内容，并在分页模式下响应时钟中断。下面我们将分析到底发生了什么事情。

【背景】探测计算机系统中的物理内存分布和大小

在proj5中，操作系统需要知道了解整个计算机系统中的物理内存如何分布的，哪些被可用，哪些不可用。其基本方法是通过BIOS中断调用来帮助完成的。其中BIOS中断调用必须在实模式下进行，所以在bootloader进入保护模式前完成这部分工作相对比较合适。这些部分由boot/bootasm.S中从probe_memory处到finish_probe处的代码部分完成完成。通过BIOS中断获取内存可调用参数为e820h的INT 15h BIOS中断。BIOS通过系统内存映射地址描述符（Address Range Descriptor）格式来表示系统物理内存布局，其具体表示如下：

```
Offset  Size    Description
00h    8字节   base address               #系统内存块基地址
08h    8字节   length in bytes            #系统内存大小
10h    4字节   type of address range     #内存类型
```

看下面的(Values for System Memory Map address type) Values for System Memory Map address type:

```
01h    memory, available to OS
02h    reserved, not available (e.g. system ROM, memory-mapped device)
03h    ACPI Reclaim Memory (usable by OS after reading ACPI tables)
04h    ACPI NVS Memory (OS is required to save this memory between NVS sessions)
other  not defined yet -- treat as Reserved
```

INT15h BIOS中断的详细调用参数:

```
eax：e820h：INT 15的中断调用参数；
edx：534D4150h (即4个ASCII字符“SMAP”) ，这只是一个签名而已；
ebx：如果是第一次调用或内存区域扫描完毕，则为0。 如果不是，则存放上次调用之后的计数值；
ecx：保存地址范围描述符的内存大小,应该大于等于20字节；
es:di：指向保存地址范围描述符结构的缓冲区，BIOS把信息写入这个结构的起始地址。
```

此中断的返回值为:

```
cflags的CF位：若INT 15中断执行成功，则不置位，否则置位；
eax：534D4150h ('SMAP') ；
es:di：指向保存地址范围描述符的缓冲区,此时缓冲区内的数据已由BIOS填写完毕
ebx：下一个地址范围描述符的计数地址
ecx    ：返回BIOS往ES:DI处写的地址范围描述符的字节大小
ah：失败时保存出错代码
```

这样，我们通过调用INT 15h BIOS中断，递增di的值（20的倍数），让BIOS帮我们查找出一个一个的内存布局entry，并放入到一个保存地址范围描述符结构的缓冲区中，供后续的ucore进一步进行物理内存管理。这个缓冲区结构定义在memlayout.h中：

```
struct e820map {
    int nr_map;
    struct {
        long long addr;
        long long size;
        long type;
    } map[E820MAX];
};
```

【实现】物理内存探测

物理内存探测是在bootasm.S中实现的，相关代码很短，如下所示：

```
probe_memory:
//对0x8000处的32位单元清零,即给位于0x8000处的
//struct e820map的结构域nr_map清零
    movl $0, 0x8000
    xorl %ebx, %ebx
//表示设置调用INT 15h BIOS中断后，BIOS返回的映射地址描述符的起始地址
    movw $0x8004, %di
start_probe:
    movl $0xE820, %eax // INT 15的中断调用参数
//设置地址范围描述符的大小为20字节，其大小等于struct e820map的结构域map的大小
    movl $20, %ecx
//设置edx为534D4150h (即4个ASCII字符“SMAP”)，这是一个约定
    movl $SMAP, %edx
//调用int 0x15中断，要求BIOS返回一个用地址范围描述符表示的内存段信息
    int $0x15
//如果eflags的CF位为0，则表示还有内存段需要探测
    jnc cont
//探测有问题，结束探测
    movw $12345, 0x8000
    jmp finish_probe
cont:
//设置下一个BIOS返回的映射地址描述符的起始地址
    addw $20, %di
//递增struct e820map的结构域nr_map
    incl 0x8000
//如果INT0x15返回的ebx为零，表示探测结束，否则继续探测
    cmpl $0, %ebx
    jnz start_probe
finish_probe:
```

上述代码正常执行完毕后，在0x8000地址处保存了从BIOS中获得的内存分布信息，此信息按照struct e820map的设置来进行填充。这部分信息将在bootloader启动ucore后，由ucore的page_init函数来根据struct e820map的memmap（定义了起始地址为0x8000）来完成对整个机器中的物理内存的总体管理。

【原理】分页内存管理

在分页内存管理中，一方面把实际物理内存（也称主存）划分为许多个固定大小的内存块，称为物理页面，或者是页框（page frame）；另一方面又把CPU（包括程序员）看到的虚拟地址空间也划分为大小相同的块，称为虚拟页面，或者简称为页面、页（page）。页面的大小要求是2的整数次幂，一般在256个字节到4M字节之间。在本书中，页面的大小设定为4KB。在32位的86x86中，虚拟地址空间是4GB，物理地址空间也也是4GB，因此在理论上程序可访问到1M个虚拟页面和1M个物理页面。软件的每一物理页面都可以放置在主存中的任何地方，分页系统（需要CPU等硬件系统提供相应的分页机制硬件支持，详见下一节）提供了程序中使用的虚地址和主存中的物理地址之间的动态映射。这样当程序访问一个虚拟地址时，支持分页机制的相关硬件自动把CPU访问的虚拟地址虚拟地址拆分为页号（可能有多级页号）和页内偏移量，再把页号映射为页帧号，最后加上页内偏移组成一个物理地址，这样最终完成对这个地址的读/写/执行等操作。

假设程序在运行时要去读地址0x100的内容到寄存器1（用REG1表示）中，执行如下的指令：

```
mov 0x100, REG1
```

虚拟地址0x100被发送给CPU内部的内存管理单元（MMU），然后MMU通过支持分页机制的相关硬件逻辑就会把这个虚拟地址是位于第0个虚拟页面当中（设页大小为4KB），页内偏移是0x100；而操作系统的分页管理子系统已经设置好第0个虚拟页面对应的是第2个物理页帧，物理页帧的起始地址是0x2000，然后再加上页内的偏移地址0x100，所以最后得到的物理地址就是0x2100。然后MMU就会把这个真正的物理地址发送到计算机系统中的地址总线上，从而可正确访问相应的物理内存单元。

如果操作系统的分页管理子系统没有设置第0个虚拟页面对应的物理页帧，则表示第0个虚拟页面当前没有对应的物理页帧，这会导致CPU产生一个缺页异常，由操作系统的缺页处理服务例程来选择如何处理。如果缺页处理服务例程认为这是一次非法访问，它将报错，终止软件运行；如果它认为是一次合理的访问，则它会采用分配物理页等手段建立正确的页映射，使得能够重新正确执行产生异常的访存指令。

【背景】X86的分页硬件支持

X86 CPU对实际物理内存的访问是通过连接着CPU和北桥芯片的前端总线来完成的，在前端总线上传输的内存地址是物理内存地址。物理内存地址被北桥映射到实际的内存条中的内存单元相应位置上。然而，在CPU内执行带来软件所使用的是虚拟内存地址（也称逻辑内存地址），它必须被转换成物理地址后，才能用于实际内存访问。

前面已经讲过了80x86的分段机制，80x86的分页机制建立在其分段机制基础之上，提供了更加强大的内存管理支持。需要注意的是，在x86中，必须先有分段机制，才能有分页机制。在分段机制中，虚地址会转换为线性地址。如果不启动分页机制，那么线性地址就是最终在前端总线上的物理地址；如果启动了分页机制，则线性地址还会经过页映射被转换为物理地址。

那如果启动分页机制呢？在80x86中有一个CR0控制寄存器，它包含一个PG位，如果PG=1，启用分页机制；如果 PG=0，禁用分页机制。不像分段机制管理大小不固定的内存卡，分页机制以固定大小的存储块为最小管理单位，即把整个地址空间（包括线性地址和物理地址）都看成由固定大小的存储块组成。在80x86中，这个固定大小一般设定为4096字节。在线性地址空间中的最小管理单位（称为页（page）），可以映射到物理地址空间中的任何一个最小管理单位（称为页帧（page frame））。页/页帧的32位地址由20位的页号/页帧号和12位的页/页帧内偏移组成。

80x86分页机制中的分页转换功能（即线性地址到物理地址的映射功能）需采用驻留在内存中的数组来描述，该数组称为页表（page table）。每个数组项就是一个页表项。由于页/页帧基地址按4096字节对齐，因此页/页帧的基地址的低12位是0。页地址<->页帧地址的转换过程以简单地看做80x86对页表的一个查找过程。页地址（线性地址）的高20位（即页号，or页的基地址）构成这个数组的索引值，用于选择对应页帧的页帧号（即页帧的基地址）。页地址的低12位给出了页内偏移量，加上对应的页帧基地址就最终形成对应的页帧地址（即物理地址）。

由于80x86的地址空间可达到4GB，按页大小（4KB）划分为1M个页。如果用一个页表来描述这种映射，那么该也表就要有1M个表项，若每个表项占用4个字节，那么该映射表就要占用4M字节。考虑到将来一个进程就需要一个地址映射表，若有多个进程，那地址映射表所占的总空间将非常巨大。为避免地址映射表占用过多的内存资源，80x86把地址映射表设定为两级。地址映射表的第一级称为页目录表，存储在一个4KB的物理页中，页目录表共有1K个表项，其中每个表项为4字节长，页表项中包含对应第二级表所在的基地址。地址映射表的第二级称为页表，每个页表也安排在一个4K字节的页中，每张页表中有1K个表项，每个表项为4字节长，包含对应页帧的基地址。由于页目录表和页表均由1K个表项组成，所以使用10位的索引就能指定表项，即用10位的索引值乘以4加基地址就得到了表项的物理地址。按上述的地址转换描述，一个页表项只需20位，但实际的页表项是32位，那其他的12位有何用途呢？

在80x86中的的页目录表项结构定义如下所示：

	7	6	5	4	3	2	1	0
7~0位	PSE	0	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页表地址				OS专用			0
23~16位	11~4位页表地址							
31~24位	19~12位页表地址							

在80x86中的的页表项结构定义如下所示：

	7	6	5	4	3	2	1	0
7~0位	0	D	A	PCD	PWT	U/S	R/W	P
15~8位	3~0位页面地址				OS专用			0
23~16位	11~4位页面地址							
31~24位	19~12位页面地址							

其中低12位的相应属性位含义如下：

- P位：存在（Present）标志，用于指明此表项是否有效。P=1表示有效；P=0表示无效。如果80x86访问一个无效的表项，则会产生一个异常。如果P=0，那么除表示表项无效外，其余位用于其他用途（比如swap in/out中，用来保存已存储在磁盘上的页面的序号）。
- R/W：读/写（Read/Write）标志，如果R/W=1，表示页的内容可以被读、写或执行。如果R/W=0，表示页的内容只读或可执行。当处理器运行在特权级（级别0、1或2）时，则R/W位不起作用。
- U/S：是用户态/特权态（User/Supervisor）标志。如果U/S=1，那么在用户态和特权态都可以访问该页。如果U/S=0，那么只能在特权态（0、1或2）可访问该页。
- A：是已访问（Accessed）标志。当CPU访问页表项映射的物理页时，页表项的这个标志就会被置为1。可通过软件把该标志位清零，并且操作系统可通过该标志来统计页的使用情况，用于页替换策略。
- D：是页面已被修改（Dirty）标志。当CPU写页表项映射的物理页内容时，页表项的这个标志就会被置为1。可通过软件把该标志位清零，并且操作系统可通过该标志来统计页的修改情况，用于页替换策略。

下图显示了由页目录表和页表构成的二级页表映射架构。

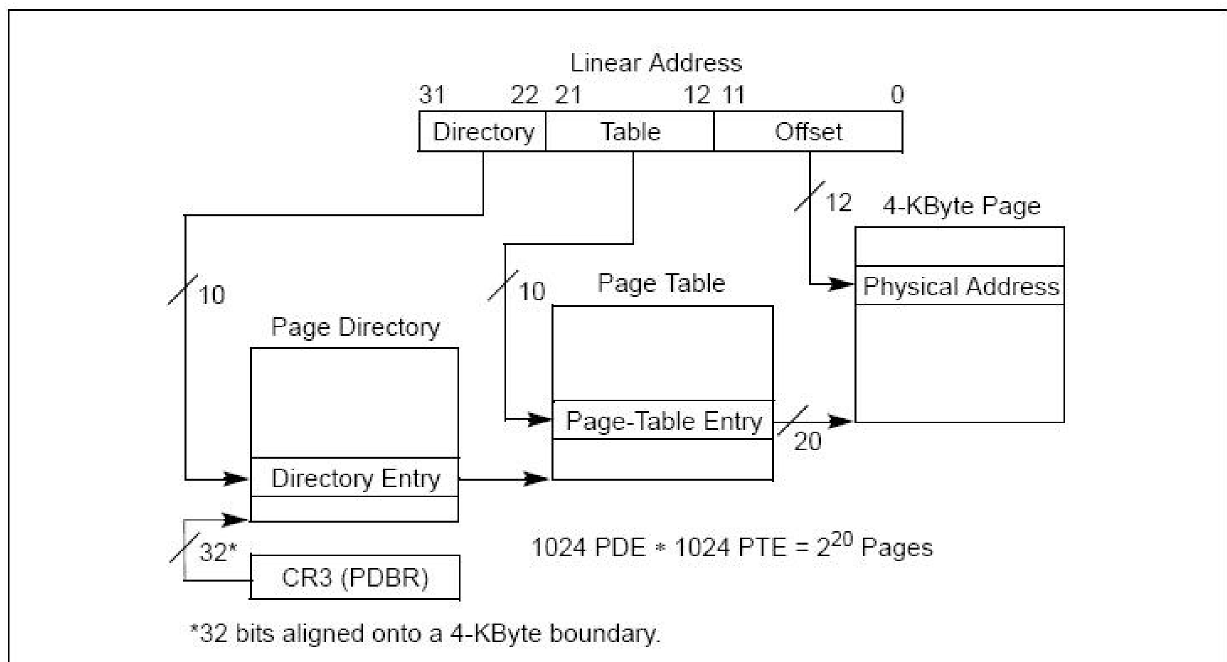


图 页目录表和页表构成的二级页表映射架构

从图中可见，控制寄存器CR3的内容是对应页目录表的物理基地址；页目录表可以指定1K个页表，这些页表可以分散存放在任意的物理页中，而不需要连续存放；每张页表可以指定1K个任意物理地址空间的页。存储页目录表和页表的基地址是按4KB对齐。当采用上述页表结构后，基于分页的线性地址到物理地址的转换过程如下图所示：

首先，CPU把控制寄存器CR3的高20位作为页目录表所在物理页的物理基地址，再把需要进行地址转换的线性地址的最高10位(即22~ 31位)作为页目录表的索引，查找到对应的页目录表项，这个表项中所包含的高20位是对应的页表所在物理页的物理基地址；然后，再把线性地址的中间10位(即12~21位)作为页表中的页表项索引，查找到对应的页表项，这个表项所包含的的高20位作为线性地址的基地址（即页号）对应的物理地址的基地址（即页帧号）；最后，把页帧号作为32位物理地址的高20位，把线性地址的低12位不加改变地作为32位物理地址的低12位，形成最终的物理地址。

如果每次访问内存单元都要访问位于内存中的页表，则访存开销太大。为了避免这类开销，x86 CPU把最近使用的地址映射数据存储在其内部的页转换高速缓存（页转换查找缓存，简称TLB）中。这样在访问存储器页表之前总是先查阅高速缓存，仅当必须的转换不在高速缓存中时，才访问存储器中的两级页表。

【实现】实现分页内存管理

重新建立段映射

前面已经介绍了如何探测物理内存，接下来ucore需要根据物理内存的情况来建立分页管理机制。首先观察一下tools/kernel.ld文件在proj4.1和proj5中的区别，在proj4.1中：

```
ENTRY(kern_init)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0x100000;

    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

在porj5中：

```
ENTRY(kern_entry)

SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0xC0100000;

    .text : {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }
```

在意味着gcc编译出ucore的起始地址从0xC0100000开始，入口函数为kern_entry函数。这与proj4.1有很大差别。这实际上说明ucore在建立好页映射关系后，虚拟地址空间和物理地址空间之间存在如下的映射关系：

```
Virtual Address=LinearAddress=0xC0000000+Physical Address
```

另外，ucore的入口地址也改为了kern_entry函数，这个函数位于init/entry.S中，分析代码可以看出，entry.S重新建立了段映射关系，从以前的

```
Virtual Address= Linear Address
```

改为

```
Virtual Address=Linear Address-0xC0000000
```

由于gcc编译出的虚拟起始地址从0xC0100000开始，ucore被bootloader放置在从物理地址0x100000处开始的物理内存中。所以当kern_entry函数完成新的段映射关系后，且ucore在没有建立好页映射机制前，CPU按照ucore中的虚拟地址执行，能够被分段机制映射到正确的物理地址上，确保ucore运行正确。

初始化物理内存页分配管理

为了与以后的分页机制配合，我们首先需要建立对整个计算机的页级物理内存分配管理。这部分代码的实现在kern/default_pmm.[ch]。首先我们需要用一个数据结构来描述每个物理页（也称页帧），这里用了双向链表结构来表示每个页。链表头用free_area_t结构来表示，包含了一个list_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr_free。

```
/* free_area_t - maintains a doubly linked list to record free (unused) pages */
typedef struct {
    list_entry_t free_list;            // the list header
    unsigned int nr_free;            // # of free pages in this free list
} free_area_t;
```

每一个物理页的属性用结构Page来表示，它包含了映射此物理页的虚拟页个数，描述物理页属性的flags和双向链接各个Page结构的page_link双向链表。

```
struct Page {
    atomic_t ref;    // page frame's reference counter
    uint32_t flags;    // array of flags that describe the status of the page frame
    list_entry_t page_link;    // free list link
};
```

有了这两个数据结构，ucore就可以管理起来整个以页为单位的物理内存空间。接下来需要解决两个问题：

- 管理页级物理内存空间所需的Page结构的内存空间从哪里开始，占多大空间？
- 空闲内存空间的起始地址在哪里？

对于这两个问题，我们首先根据bootloader给出的内存布局信息找出最大的物理内存地址maxpa（定义在page_init函数中的局部变量），由于x86的起始物理内存地址为0，所以可以得知需要管理的物理页个数为

```
npage = maxpa / PGSIZE
```

这样，我们就可以预估出管理页级物理内存空间所需的Page结构的内存空间所需的内存大小为：

```
sizeof(struct Page) * npage)
```

由于bootloader加载ucore的结束地址（用全局指针变量end记录）以上的空间没有被使用，所以我们可以把end按页大小为边界去整后，作为管理页级物理内存空间所需的Page结构的内存空间，记为：

```
pages = (struct Page *)ROUNDUP((void *)end, PGSIZE);
```

为了简化起见，从地址0到地址pages+ sizeof(struct Page) *npage)*结束的物理内存空间设定为已占用物理内存空间（起始*0~640KB*的空间是空闲的），地址*pages+ sizeof(struct Page)* npage)以上的空间为空闲物理内存空间，这时的空闲空间起始地址为

```
uintptr_t freemem = PADDR((uintptr_t)pages + sizeof(struct Page) * npage);
```

为此我们需要把这两部分空间给标识出来。对于已占用物理空间，通过如下语句即可实现占用标记：

```
for (i = 0; i < npage; i ++) {
    SetPageReserved(pages + i);
}
```

对于空闲物理空间，通过如下语句即可实现空闲标记：

```
//获得空闲空间的起始地址begin和结束地址end
……
init_memmap(pa2page(begin), (end - begin) / PGSIZE);
```

其实SetPageReserved只需把物理地址对应的Page结构中的flags标志设置为PG_reserved ，表示这些页已经被使用了。而init_memmap函数则是把空闲物理页对应的Page结构中的flags和引用计数ref清零，并加到free_area.free_list指向的双向列表中，为将来的空闲页管理做好初始化准备工作。

物理内存页分配与释放

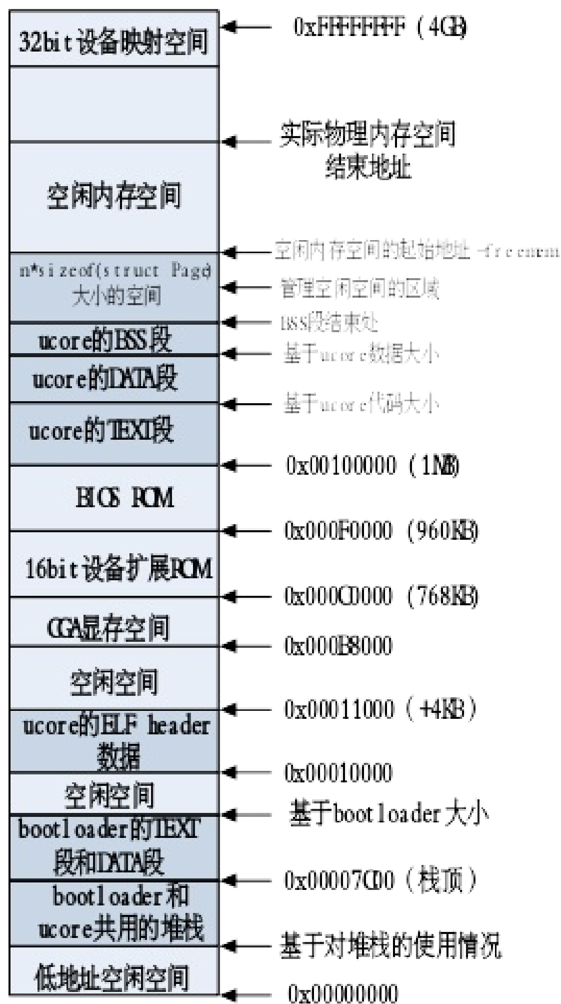
关于内存分配的操作系统原理方面的知识有很多，但在proj5中只实现了最简单的内存页分配算法，即每次只分配一页或释放一页的内存页分配算法。相应的实现在default_pmm.c中的default_alloc_pages函数和default_free_pages函数，相关实现很简单，这里就不具体分析

了，直接看源码，应该很好理解。

其实proj5在内存分配和释放方面最主要的作用是建立了一个物理内存页管理器框架，这实际上是一个函数指针列表，定义如下：

```
struct pmm_manager {
    const char *name; //物理内存页管理器的名字
    void (*init)(void); //初始化内存管理器
    void (*init_memmap)(struct Page *base, size_t n); //初始化管理空闲内存页的数据结构
    struct Page *(*alloc_pages)(size_t n); //分配n个物理内存页
    void (*free_pages)(struct Page *base, size_t n); //释放n个物理内存页
    size_t (*nr_free_pages)(void); //返回当前剩余的空闲页数
    void (*check)(void); //用于检测分配/释放实现是否正确的辅助函数
};
```

重点是实现init_memmap/ alloc_pages/ free_pages这三个函数。当完成物理内存页管理初始化工作后，计算机系统的内存布局如下图所示：



`chapt3-proj5-memory.vsd`

读者可进一步通过分析proj5.1/5.1.1/5.1.2/5.2中firstfit_pmm[ch]/bestfit_pmm[ch]/worstfit_pmm[ch]/ buddy_pmm[ch]文件中对应函数实现来体会原理课中的连续空间内存分配中各种分配算法的设计思路和实现。

建立二级页表

为了实现分页机制，需要建立好虚拟内存和物理内存的页映射关系，即建立二级页表。这需要解决如下问题：

- 对于哪些物理内存空间需要建立页映射关系？
- 具体的页映射关系是什么？
- 页目录表的起始地址设置在哪里？
- 页表的起始地址设置在哪里，需要多大空间？
- 如何设置页目录表项的内容？
- 如何设置页目录表项的内容？

下面我们逐一解决上述问题。由于物理内存页管理器管理了从0到实际可用物理内存大小的物理内存空间，所以对于这些物理内存空间都需要建立好页映射关系。由于目前ucore只运行在内核空间，所以可以建立一个一一映射关系。假定虚拟内核地址的起始地址为0xC0000000，这虚拟内存和物理内存的具体页映射关系为：

```
Virtual Address=Physical Address+0xC0000000
```

由于我们已经具有了一个物理内存页管理器default_pmm_manager，我们就可以用它来获得所需的空闲物理页。在二级页表结构中，页目录表占4KB空间，ucore就可通过default_pmm_manager的default_alloc_pages函数获得一个空闲物理页，这个页的起始物理地址就是页目录表的起始地址。同理，ucore也通过这种方式获得各个页表所需的空间。页表的空间大小取决与页表要管理的物理页数n，一个页表项（32位，即4字节）可管理一个物理页，页表需要占n/256个物理页空间。这样页目录表和页表所占的总大小为4096+1024*n字节。

为把0~KERNSIZE（明确ucore设定实际物理内存不能超过KERNSIZE值，即0x38000000字节，896MB，3670016个物理页）的物理地址一一映射到页目录表项和页表项的内容，其大致流程如下：

1. 先通过default_pmm_manager获得一个空闲物理页，用于页目录表；
2. 调用boot_map_segment函数建立一一映射关系，具体处理过程以页为单位进行设置，即

    ```
    Virtual Address=Physical Address+0xC0000000
    ```

  - 设一个逻辑地址la（按页对齐，故低12位为零）对应的物理地址pa（按页对齐，故低12位为零），如果在页目录表项（la的高10位为索引值）中的存在位（PTE_P）为0，表示缺少对应的页表空间，则可通过default_pmm_manager获得一个空闲物理页给页表，页表起始物理地址是按4096字节对齐的，这样填写页目录表项的内容为

    页目录表项内容 = 页表起始物理地址| PTE_U | PTE_W | PTE_P

  - 进一步对于页表中对应页表项（la的中10位为索引值）的内容为

    ```
    页表项内容 =  pa | PTE_P | PTE_W
    ```

    其中：

    - PTE_U：位3，表示用户态的软件可以读取对应地址的物理内存页内容
    - PTE_W：位2，表示物理内存页内容可写
    - PTE_P：位1，表示物理内存页存在

建立好一一映射的二级页表结构后，接下来就要使能分页机制了，这主要是通过enable_paging函数实现的，这个函数主要做了两件事：

- 通过lcr3指令把页目录表的起始地址存入CR3寄存器中；
- 通过lcr0指令把cr0中的CR0_PG标志位设置上。

执行完enable_paging函数后，计算机系统进入了分页模式！但到这一步还不够，还记得ucore在最开始通过kern_entry函数设置了临时的新段映射机制吗？这个临时的新段映射机制不是最简单的对等映射，导致虚拟地址和线性地址不相等。而刚才建立的页映射关系是建立在简单的段对等映射，即虚拟地址=线性地址的假设基础之上的。所以我们需要进一步调整段映射关系，即重新设置新的GDT，建立对等段映射。

这里需要注意：在进入分页模式到重新设置新GDT的过程是一个过渡过程。在这个过渡过程中，已经建立了页表机制，所以通过现在的段机制和页机制实现的地址映射关系为：

```
Virtual Address=Linear Address + 0xC0000000 = Physical Address +0xC0000000+0xC0000000
```

在这个特殊的阶段，如果不把段映射关系改为Virtual Address = Linear Address，则通过段页式两次地址转换后，无法得到正确的物理地址。为此我们需要进一步调用gdt_init函数，根据新的gdt全局段描述符表内容（gdt定义位于pmm.c中），恢复以前的段映射关系，即使得Virtual Address = Linear Address。这样在执行完gdt_init后，通过的段机制和页机制实现的地址映射关系为：

```
Virtual Address=Linear Address = Physical Address +0xC0000000
```

这里存在的一个问题是，在调用enable_page函数使能分页机制后到执行完毕gdt_init函数重新建立好段页式映射机制的过程中，内核使用的还是旧的段表映射，也就是说，enable paging 之后，内核使用的是页表的低地址 entry。 如何保证此时内核依然能够正常工作呢？其实只需让低地址目录表项的内容等于以KERNBASE开始的高地址目录表项的内容即可。目前内核大小不超过 4M （实际上是3M，因为内核从 0x100000 开始编址），这样就只需要让页表在0~4MB的线性地址与KERNBASE ~ KERNBASE+4MB的线性地址获得相同的映射即可，都映射到 0~4MB 的物理地址空间，具体实现在pmm.c中pmm_init函数的语句：

```
boot_pgdir[0] = boot_pgdir[PDX(KERNBASE)];
```

实际上这种映射也限制了内核的大小。当内核大小超过预期的3MB 就可能导致打开分页之后内核 crash，在后面的试验中，也的确出现了这种情况。解决方法同样简单，就是拷贝更多的高地址项到低地址。

当执行完毕gdt_init函数后，新的段页式映射已经建立好了，上面的0~4MB的线性地址与0~4MB的物理地址一一映射关系已经没有用了。所以可以通过如下语句解除这个老的映射关系。

```
boot_pgdir[0] = 0;
```

自映射机制

上一小节讲述了通过boot_map_segment函数建立了基于一一映射关系的页目录表项和页表项，这里的映射关系为：

```
Virtual addr (KERNBASE~KERNBASE+KMEMSIZE) = Physical_addr (0~KMEMSIZE)
```

这样只要给出一个虚地址和一个物理地址，就可以设置相应PDE和PTE，就可完成正确的映射关系。

如果我们这时需要按虚拟地址的地址顺序显示整个页目录表和页表的内容，则要查找页目录表的页目录表项内容，根据页目录表项内容找到页表的物理地址，再转换成对应的虚地址，然后访问页表的虚地址，搜索整个页表的每个页目录项。这样过程比较繁琐。我们有没有一个简洁的方法来实现这个查找呢？ucore做了一个很巧妙的地址自映射设计，把页目录表和页表放在一个连续的4MB虚拟地址空间中，并设置页目录表自身的虚地址<-->物理地址映射关系。这样在已知页目录表起始虚地址的情况下，通过连续扫描这特定的4MB虚拟地址空间，就很容易访问每个页目录表项和页表项内容。

具体而言，ucore是这样设计的，首先设置了一个常量（memlayout.h）：

```
VPT=0xFAC00000，
```

这个地址的二进制表示为：

```
1111 1010 1100 0000 0000 0000 0000 0000
```

高10位为1111 1010 11，即10进制的1003，中间10位为0，低12位也为0。在pmm.c中有两个全局初始化变量

```
pte_t * const vpt = (pte_t *)VPT;
pde_t * const vpd = (pde_t *)PGADDR(PDX(VPT), PDX(VPT), 0);
```

NaN. 并在pmm_init函数执行了如下语句：

```
boot_pgdir[PDX(VPT)] = PADDR(boot_pgdir) | PTE_P | PTE_W;
```

这些变量和语句有何特殊含义呢？其实vpd变量的值就是页目录表的起始虚地址0xFAFEB000，且它的高10位和中10位是相等的，都是10进制的1003。当执行了上述语句，就确保了vpd变量的值就是页目录表的起始虚地址，且vpt是页目录表中第一个目录表项指向的页表的起始虚地址。此时描述内核虚拟空间的页目录表的虚地址为0xFAFEB000，大小为4KB。页表的理论连续虚拟地址空间0xFAC00000~0xFB000000，大小为4MB。因为这个连续地址空间的大小为4MB，可有1M个PTE，即可映射4GB的地址空间。

但ucore实际上不会用完这么多项，在memlayout.h中定义了常量

```
#define KMEMSIZE            0x38000000
```

表示ucore只支持896MB的物理内存空间，这个896MB只是一个设定，可以根据情况改变。则最大的内核虚地址为常量

```
\#define KERNTOP            (KERNBASE + KMEMSIZE)=0xF8000000
```

所以最大内核虚地址KERNTOP的页目录项虚地址为

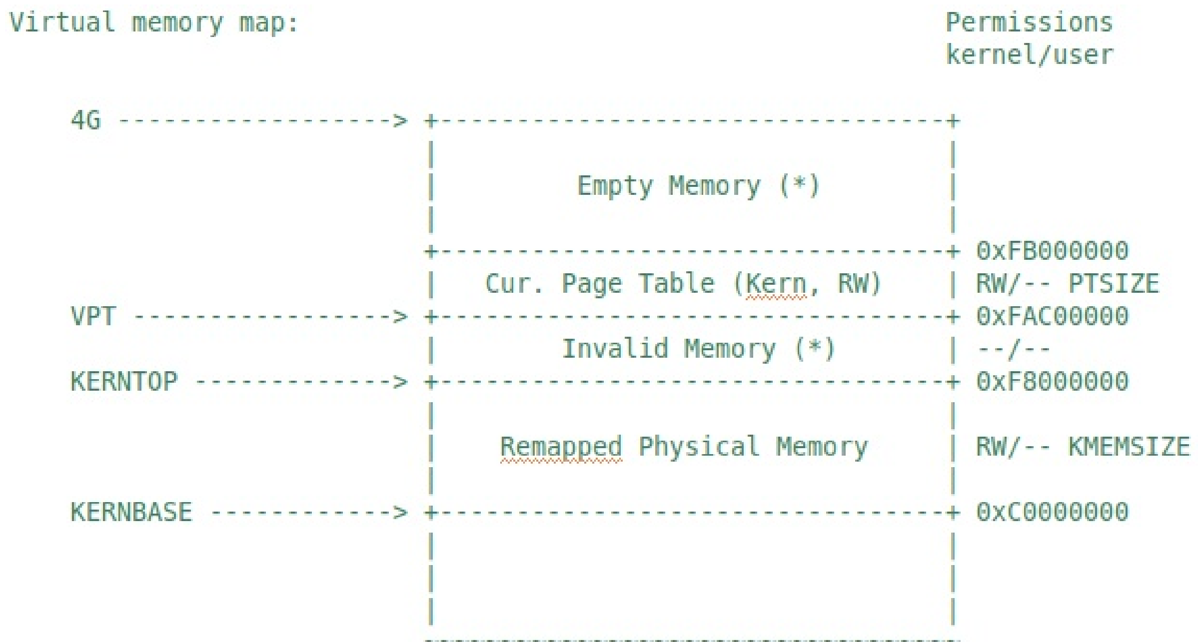
```
vpd+0xF8000000/0x400000=0xFAFEB000+0x3E0=0xFAFEB3E0
```

最大内核虚地址KERNTOP的页表项虚地址为：
vpt+0xF8000000/0x1000=0xFAC00000+0xF8000=0xFACF8000

在pmm.c中的函数print_pgdir就是基于ucore的页表自映射方式完成了对整个页目录表和页表的内容扫描和打印。注意，这里不会出现某个页表的虚地址与页目录表虚地址相同的情况。

自映射机制还可方便用户态程序访问页表。因为页表是内核维护的，用户程序很难知道自己页表的映射结构。VPT 实际上在内核地址空间的，我们可以用同样的方式实现一个用户地址空间的映射（比如 pgdir[UVPT] = PADDR(pgdir) | PTE_P | PTE_U，注意，这里不能给写权限，并且 pgdir 是每个进程的 page table，不是 boot_pgdir），这样，用户程序就可以用和内核一样的 print_pgdir 函数遍历自己的页表结构了。

在page_init函数建立完实现物理内存一一映射和页目录表自映射的页目录表和页表后，一旦使能分页机制，则ucore看到的内核虚拟地址空间如下图所示：



proj5使能分页机制后的虚拟地址空间图

【原理】页内存分配算法

在proj5中进行在动态分配内存时，存在很多限制，效率很低。在操作系统原理中，为了有效地分配内存，首先需要了解和跟踪空闲内存和分布情况，一般可采用位图（bit map）和双向链表两种方式跟踪内存使用情况。若采用位图方式，则每个页对应位图区域的一个bit，如果此位为0，表示空闲，如果为1，表示被占用。采用位图方式很省空间，但查找n个长度为0的位串的开销比较大。而双向链表在查询或修改操作方面灵活性和效率较高，所以ucore采用双向链表来跟踪跟踪内存使用情况。

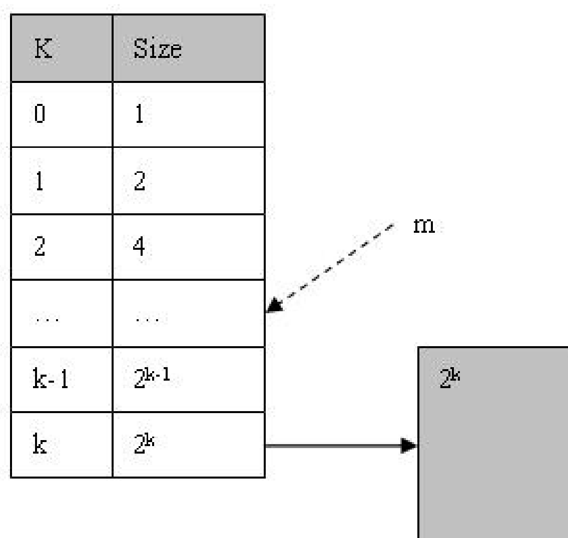
假设整个物理内存空闲空间的以页为单位被一个双向链表管理起来，每个表项管理一个物理页。这需要设计某种算法来查找空闲页和回收空闲页。ucore实现了首次适配（first fit）算法、最佳适配（best fit）算法、最差适配（worst fit）算法和兄弟（buddy）算法，这些算法都可以实现在ucore提供的物理内存页管理器框架pmm_manager下。

首次适配（first fit）算法的分配内存的设计思路是物理内存页管理器顺着双向链表进行搜索空闲内存区域，直到找到一个足够大的空闲区域，这是一种速度很快的算法，因为它尽可能少地搜索链表。如果空闲区域的大小和申请分配的大小正好一样，则把这个空闲区域分配出去，成功返回；否则将该空闲区分为两部分，一部分区域与申请分配的大小相等，把它分配出去，剩下的一部分区域形成新的空闲区。其释放内存的设计思路很简单，只需把这块区域重新放回双向链表中即可。

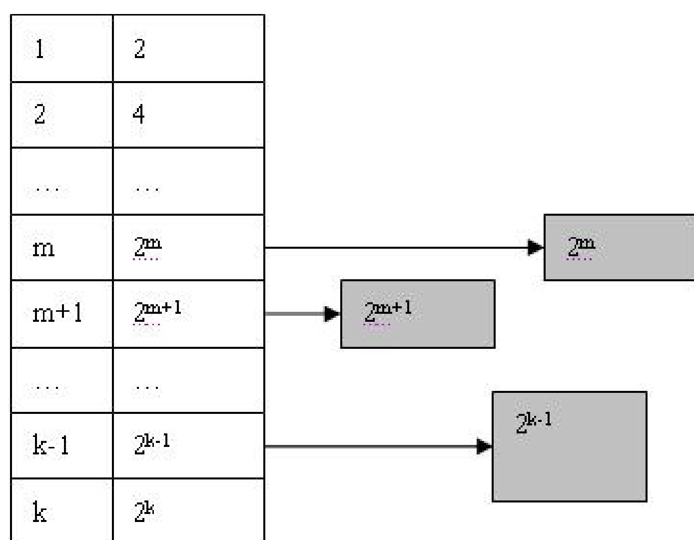
最佳适配（best fit）算法的设计思路是物理内存页管理器搜索整个双向链表（从开始到结束），找出能够满足申请分配的空间大小的最小空闲区域。找到这个区域后的处理以及释放内存的处理与上面类似。最佳适配算法试图找出最接近实际需要的空闲区，名字上听起来很好，其实在查询速度上较慢，且较易产生多的内存碎片。

最差适配（worst fit）算法与最佳适配（best fit）算法的设计思路相反，物理内存页管理器搜索整个双向链表，找出能够满足申请分配的空间大小的最大空闲区域，使新的空闲区比较大从而可以继续使用。在实际效果上，查询速度上也较慢，产生内存碎片相对少些。

上述三种算法在实际应用中都会产生碎片较多，效率不高的问题。为此一般操作系统会采用buddy算法来改进上述问题。buddy算法的基本设计思想是：在buddy系统中，被占用的内存空间和空闲内存空间的大小均为2的k次幂（k是正整数）。这样在ucore中，若申请n个页的内存空间，则实际可能分配的空间大小为2K个页（2k-1\<n<=2k）。若初始化时的空闲内存空间容量为2m个页，这空闲块的大小只可能是20、21、…、2m个页。

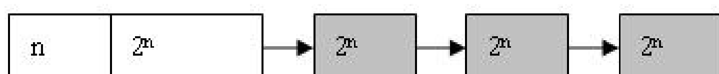


假定内存一开始是一个连续地址空间（大小为2^k个页）的大空闲块，且最小分配单位为1个页（4KB），则buddy system初始化时将生成一个长度为k + 1的可用空间表List, 并将全部可用空间作为一个大小为2^k个页的空闲块Bk挂接在空闲块数组链表List的最后一个节点上, 如下图：



当ucore其他子系统申请n个字节的存储空间时, buddy system分配的空闲块大小为2^ m个页，m满足条件：2^ (m-1) < n <= 2^ m

此时buddy system将在list中的m位置寻找可用的空闲块。初始化时List中这个位置为空, 于是buddy system就向上查找m+1，…，直到达到k位置为止. 找到k位置后, 便得到可用空闲块Bk, 此时Bk将分裂成两个大小为2^(k-1)的空闲块Bk-1a和Bk-1b, 并将其中一个插入到List中k-1位置, 同时对另外一个继续进行分裂. 如此以往直到得到两个大小为2^m个页的块为止,并把其中一个空闲块分配给需求方。此时的内存如下图所示:



如果buddy system在运行一段时间之后, List中某个位置t可能会出现多个块, 则将其他块依次链接可用块链表的末尾。当buddy system要在t位置取可用块时, 直接从链表头取一个即可。

当一个存储块被释放时, buddy system将把此内存块回收到空闲块链表List中。此时buddy system系统将根据此存储块的大小计算出其在List中的位置, 然后插入到空闲块链表的末尾。在这一步完成后, 系统立即开始合并尝试操作， 该操作是将地址相邻且大小相等的空闲块（简称buddy，即"伙伴"空闲块）合并到一起, 形成一个更大的空闲块，并重新放到空闲块链表List的对应位置中, 并继续对更大的块进行合并, 直到无法合并为止。

严蔚敏老师的“数据结构”一书第8章第4节对buddy算法有详尽的解释，“understanding linux kernel”此书对此也有很好的描述，读者可以进一步参考。

对于上述4个内存分配算法，可参考对应的proj5.1/5.1.1/5.1.2/5.2中的kern/mm/*_pmm.[ch]的具体实现来进一步了解。

```
（可以进一步描述三种算法的具体实现）
```

支持任意大小内存分配

试验目标

上一节描述了如何进行页级内存的分配与释放管理，可以比较有效地完成以页大小为最小单位（粒度）的内存分配和回收工作，这样可以很好地与分页机制配合在一起提供有效的分页管理。但在操作系统的实际运行过程中，还有很多对小于一页的任意大小内存的动态申请需求，则以页为最小单位就无法适应这类需求了。当然，我们可以直接把物理内存页管理器改为粒度为1字节的物理内存管理器，这主要存在两个问题：

- 由于页目录表和页表的大小是4KB，且一个页表项管理的内存空间大小也是4KB，故需要扩展新的函数和结构匹配对页表的管理支持，导致代码复杂；
- 即使采用上述4种内存分配算法，在支持任意大小的内存分配请求上，依然存在效率不高，有外碎片和内碎片等问题。

所以，一个更加合理的办法是在物理内存页管理器的基础之上建立一个支持任意大小的内存分配管理器，形成二级内存管理，满足高效支持任意大小的内存分配请求。这就对动态内存分配管理提出了新的挑战，即花尽量少的时间完成对内存的分配和回收，且保证能够产生的内外碎片尽量小。

proj6中参考Jeff Bonwick 为 Sun OS 操作系统首次引入的一种算法：slab算法。slab算法的基本思路有两个，一个是通过缓存实现“对象”重用，另一个是在一个连续页空间放同样类型的“对象”。

Jeff Bonwick在SUN OS内核中观察到内核在运行时会为有限的对象集（内核中各种常见的数据结构）分配大量内存，且对内核中这些数据结构进行初始化所需的时间超过了对其进行分配和释放所需的时间。因此他的结论是不应该将内存释放回一个全局的空闲内存池，而是将内存保持为针对特定目而初始化的状态。例如，如果内存被分配给了一个变量，那么只需在为此变量首次分配内存时执行一次变量初始化函数即可，当该变量被回收并进一步被后续分配时就不需要执行这个初始化函数，因为从上次释放和调用析构之后，它已经处于所需的状态中了。

在一个连续地址空间放同样类型的“对象”有助于快速查找和修改同样类型的“对象”，提高分配的时间效率，并减少碎片。

proj6概述

实现描述

proj6基于proj5.2实现，主要在buddy物理内存页管理器的基础上，增加了一级任意大小内存分配管理，通过slab算法实现对小内存的简洁分配，为后续的运行时动态内存管理提供通用的内存申请和释放接口、在proj6中，可以了解到：

- slab算法的数据结构和具体实现；
- 小内存分配管理器与物理内存页管理器的接口和交互过程。

项目组成

```
proj6
├── boot
│   └── ……
├── kern
│   ├── debug
│   │   └── ……
│   ├── driver
│   │   └── ……
│   ├── init
│   │   └── ……
│   ├── libs
│   │   └── ……
│   ├── mm
│   │   ├── ……
│   │   ├── memlayout.h
│   │   ├── slab.c
│   │   └── slab.h
│   ├── sync
│   │   └── ……
│   └── trap
│       └── ……
├── libs
│   └── ……
├── Makefile
└── ……

11 directories, 58 files
```

相对与proj5.2，proj6增加了slab.[ch]两个个文件，主要完成对slab内存管理算法的简单实现。

编译运行

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc0109530 (phys)
  edata  0xc0122aa0 (phys)
  end    0xc0123cb8 (phys)
Kernel executable memory footprint: 144KB
memory managment: buddy_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_slab() succeeded!
++ setup timer interrupts
100 ticks
100 ticks
```

【实现】slab算法的简化设计实现

数据结构描述

slab 算法采用了两层数据组织结构。在最高层是 slab_cache，这是一个不同大小slab 缓存的链接列表数组。slab_cache的每个数组元素都是一个管理和存储给定大小的空闲对象（obj）的slab 结构链表，这样每个slab设定一个要管理的给定大小的对象池，占用物理空间连续的1个或多个物理页。slab_cache的每个数组元素管理两种slab列表：

- slabs_full：完全分配的 slab
- slabs_notfull：部分分配的 slab

注意 slabs_notfull列表中的 slab 是可以进行回收（reaping），使得slab 所使用的内存可被返回给操作系统供其他子系统使用。

slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），它们被划分成一个个obj。这些obj是中进行分配和释放的基本元素。由于对象是从 slab 中进行分配和释放的，因此单个 slab 可以在 slab 链表之间进行移动。例如，当一个 slab 中的所有对象都被使用完时，就从 slabs_notfull 链表中移动到 slabs_full 链表中。当一个 slab 完全被分配并且有对象被释放后，就从 slabs_full 列表中移动到 slabs_notfull列表中。下面是ucore中的slab架构图：



```
 +----------------------------------+
 | slab_cache[0] for 0~32B obj      |
 +----------------------------------+
 | slab_cache[1] for 33B~64B obj    |-->lists for slabs
 +----------------------------------+            |
 | slab_cache[2] for 65B~128B obj   |            |
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~            |
 +----------------------------------+            |
 | slab_cache[12]for 64KB~128KB obj |            |
 +----------------------------------+            |
                                                 |
   slabs_full/slabs_not    +---------------------+
 -<-----------<----------<-+
 |           |         |
slab1       slab2     slab3...
 |
 |-------|-------|
pages1  pages2   pages3...
 |
 |
 |
 slab_t+n*bufctl_t+obj1-obj2-obj3...objn  (the size of obj is small)
 |
 OR
 |
 obj1-obj2-obj3...objn  WITH slab_t+n*bufctl_t in another slab (the size of obj is BIG)
```

slab架构图

分配与释放内存实现描述

现在来看一下能够创建新 slab 缓存、向缓存中增加内存、销毁缓存的接口以及 slab 中对对象进行分配和释放操作的slab相关操作过程和函数。

第一个步骤是通过执行slab_init函数初始化slab_cache 缓存结构。然后其他 slab 缓存函数将使用该引用进行内存分配与释放等操作。ucore中最常用的内存管理函数是 kmalloc 和 kfree 函数。这两个函数的原型如下：

- void *kmalloc(size_t size);
- void kfree(void *objp);

在分配任意大小的空闲块时，kmalloc通过调用kmem_cache_alloc函数来遍历对应size的slab，来查找可以满足大小限制的缓存。如果kmem_cache_alloc函数发现slab中有空闲的obj，则分配这个对象；如果没有空闲的obj了，则调用kmem_cache_grow函数分配包含1到多页组成的空闲slab，然后继续分配。要使用 kfree 释放对象时，通过进一步调用kmem_cache_free把分配对象返回到slab中，并标记为空闲，建立空闲obj之间的链接关系。

（可进一步详细一些）

实现虚存管理功能

试验目标

有了页表的支持，我们可以使得不同用户态运行程序的内存空间之间无法访问，达到隔离和保护的作用。但页表如何仅仅只支持这个功能就太大材小用了。我们其实还可以通过页表实现更多的功能：

- 内存共享：把两个虚拟地址空间通过页表映射到同一物理地址空间。这只需通过设置不同索引的页表项的内容一致即可。
- 提供超过物理内存大小的虚拟内存空间：这一步需要结合异常中断处理和硬盘来完成。其基本思想是在内存中放置最常用的一些数据，不常用的数据会被放到硬盘上，但给用户态的软件一种感觉，觉得这些数据都在内存中。当用户态软件访问到的数据不在内存中（暂时存放在硬盘上）的时候，这条访存指令会引发异常中断，由操作系统的异常中断处理例程进行管理。这时操作系统会分析引发异常的内存地址，能够把对应缓存在硬盘中的数据重新读入这个内存地址，并让用户态软件重新执行产生访存异常的那条指令。这些由操作系统完成的工作在用户态完全“看”不到。从用户态软件的角度看，只是操作系统给用户提供了一个超出实际物理内存大小的虚拟内存空间。
- 按需分配内存：用户态软件在运行时要求操作系统提供很大的内存，操作系统“表面上”表示满足用户需求，但在背后并没有实际分配对应的物理内存空间。等到用户态软件实际执行到对这些内存的访问时，由于没有分配对应的物理内存空间，会导致产生访存异常。操作系统的异常中断处理例程发觉这是用户态软件以前确实要求过的内存空间，则在从系统管理的空闲空间中分配一页或几页物理内存给用户态软件，并让用户态软件重新执行产生访存异常的那条指令。这些由操作系统完成的工作在用户态也完全“看”不到。但从操作系统的整体管理的角度看，这种方式在用户态软件确实需要的时候把内存分配给用户态软件，提高了内存的使用率，避免了用户态软件“圈地不用”的现象。

为了高效地完成上述三件事情，操作系统需要考虑应该把哪些不常用的内存换出到硬盘上去，这就是内存的页替换算法，常见的有LRU算法，Clock算法，二次机会法等。而在实现上，由于涉及异常处理和硬盘管理等，虚存管理在整个ucore实现中的相对复杂度是最大的。

【原理】虚拟内存管理

什么是虚拟内存？简单地说，是指程序员或CPU “需要”和直接“看到”的内存，这其实暗示了两点：1、虚拟内存单元不一定有实际的物理内存单元对应，即实际的物理内存单元可能不存在；2、如果虚拟内存单元对应有实际的物理内存单元，那二者的地址一般不是相等的。通过操作系统的某种内存管理和映射技术可建立虚拟内存与实际的物理内存的对应关系，使得程序员或CPU访问的虚拟内存地址会转换为另外一个物理内存地址。

那么这个“虚拟”的作用或意义在哪里体现呢？在操作系统中，虚拟内存其实包含多个虚拟层次，在不同的层次体现了不同的作用。首先，在有了分段或分页机制后，程序员或CPU直接“看到”的地址已经不是实际的物理地址了，这已经有一层虚拟化，我们可简称为内存地址虚拟化。有了内存地址虚拟化，我们就可以通过设置段界限或页表项来设定软件运行时的访问空间，确保软件运行不越界，完成内存访问保护的功能。

通过内存地址虚拟化，可以使得软件在没有访问某虚拟内存地址时不分配具体的物理内存，而只有在实际访问某虚拟内存地址时，操作系统再动态地分配物理内存，建立虚拟内存到物理内存的页映射关系，这种技术属于lazy load技术，简称按需分页（demand paging）。把不经常访问的数据所占的内存空间临时写到硬盘上，这样可以腾出更多的空闲内存空间给经常访问的数据；当CPU访问到不经常访问的数据时，再把这些数据从硬盘读入到内存中，这种技术称为页换入换出（page swap in/out）。两个虚拟页的数据内容相同时，可只分配一个物理页框，这样如果对两个虚拟页的访问方式是只读方式，这这两个虚拟页可共享页框，节省内存空间；如果CPU对其中之一的虚拟页进行写操作，则这两个虚拟页的数据内容会不同，需要分配一个新的物理页框，并将物理页框标记为可写，这样两个虚拟页面将映射到不同的物理页帧，确保整个内存空间的正确访问。这种技术称为写时复制（Copy On Write，简称COW）。这三种内存管理技术给了程序员更大的内存“空间”，我们称为内存空间虚拟化。

ucore在实现上述三种技术时，需要解决的一个关键问题是，何时进行请求调页/页换入换出/写时复制处理？其实，在程序的执行过程中由于某种原因（页框不存在/写只读页等）而使CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射时，CPU会产生一次缺页异常，从而需要进行相应的缺页异常服务例程。这个缺页异常处理的时机就是求调页/页换入换出/写时复制处理的执行时机，当相关处理完成后，缺页异常服务例程会返回到产生异常的指令处重新执行，使得软件可以继续正常运行下去。

proj7/8/9/9.1/9.2概述

为了实现虚存管理，首先需要能够处理缺页异常，这是需要对当前的trap处理进行扩展，并能够描述当前内核中“合法”的虚拟内存（不一定有对应的物理内存）。proj7在proj6的基础上实现了上述过程，新增加的主要工作包括：

- 描述当前“合法”的虚拟内存的数据结构vma_struct和针对vma_struct的函数操作；
- 扩展trap_dispatch函数，使得能够根据vma_struct结构的描述，正确完成对缺页的处理（即如果发现是“合法”的虚拟内存地址，则创建或修改页表项来建立与物理内存页的对应关系）。

为了提供超过物理内存大小的虚拟内存空间，需要把不常用的页换出到硬盘上，这样当访问到这些不存在的虚存页时，会产生缺页异常，可以把这些页再从硬盘拷贝回到内存中。proj8在proj7的基础上完成上述过程的实现，新增加的主要工作包括：

- 为了准备swap in/out，实现通过PIO方式读写IDE格式的硬盘；
- 建立swap相关数据结构和相关操作，确保不常用的页能够被换出（swap out）到硬盘上，并在被访问时，能够从硬盘对应的扇区中换入（swap in）到内存中;

为了实现将来不同进程（用户态程序）之间共享内存，需要对描述虚拟内存的vma_strct结构进行扩展。proj9/9.1在proj8的基础上完成上述过程的实现，新增加的主要工作包括：

- 增加shmem_node结构的描述，确保能够描述多个虚拟页映射到一个物理页的情况，并增加针对shmem_node的处理。

- 为了减少复制内存的开销，可通过实现写时复制（Copy On Write，简称COW）机制来完成，其基本思路是在只读情况下，多个虚拟页只需映射到一个物理页上，当对虚拟页进行写操作时，才真正完成对物理页的复制。在实现上需要对page的属性进行扩展，能够在发生页保护异常时，探测出是为了“写时复制”而设置的页，这样在缺页异常处理中，会完成实际的分配新页操作。proj9.2在proj9.1的基础上完成上述过程的实现，新增加的主要工作包括：

- 扩展trap_dispatch函数，使得能够根据产生异常的地址的页表项内容和此地址对应的vma中的属性描述，正确完成对的“写时复制”处理。

proj7：支持缺页异常和VMA结构

proj7项目组成

```
proj7
|   |-- init
|   |   `-- init.c
|   |-- mm

|   |   |-- pmm.c
|   |   |-- pmm.h
|   |   |-- vmm.c
|   |   `-- vmm.h
|   |-- sync
|   |   `-- sync.h
|   `-- trap
|       |-- trap.c

|-- libs
|   `-- x86.h
```

相对与proj6，proj7主要修改和增加的文件如下：

- init.c：在kern_init中增加调用初始化虚存管理函数vmm_init
- pmm.[ch]：增加pgdir_alloc_page函数，完成分配一个空闲物理页，并设置好页表项，完成正确的虚拟地址到物理地址的转换；
- trap.c：完成对缺页异常的基本操作，调用vmm.c中的do_pgfault函数完成具体的缺页处理；
- x86.h：完成对控制寄存器CR1和CR2的读操作；
- vmm.[ch]：新增的文件，主要是建立vma_struct结构，用于描述不存在的虚拟内存，并完成针对此结构的相关操作函数。

proj7编译运行

编译并运行proj7的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
chenyu@chenyu-laptop:~/oscourse/branches/testing/chyyuu/proj7$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc010ae5f (phys)
  edata  0xc0127aa0 (phys)
  end    0xc0128cbc (phys)
Kernel executable memory footprint: 164KB
memory managment: buddy_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END --------------------
check_slab() succeeded!
size of struct mm_struct is 24, size of struct vma_struct is 40
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
++ setup timer interrupts
100 ticks
100 ticks
```

通过上图，我们可以看到ucore在check_vma_struct函数中完成基于vma_struct结构的数据创建等操作，确保能够正确建立vma_struct结构，并在成功测试后打印“check_vma_struct() succeeded!”；接下来ucore创建一个描述了虚拟地址0~4K的vma_struct结构，这个0虚拟地址起始的虚拟页没有对应的物理页，所以在实际访问这个虚拟地址的时候会产生缺页异常，中断处理例程会经过如下调用：

```
vectorXXX(vectors.S)-->\__alltraps(trapentry.S)--> trap(trap.c)-->trap_dispatch(trap.c
)—
-->pgfault_handler(trap.c)-->print_pgfault(trap.c)
```

来显示出错的位置和原因“page fault at 0x00000100: K/W [no page found].”，即在内核态对虚存地址0x100处执行写操作出现了缺页异常。并进一步调用do_pgfault函数来检测时候此虚拟地址属于某个vma_struct描述的范畴，如果是，则会分配一个物理页来对应此虚拟地址所在的虚拟页，并返回继续执行引起缺页异常的指令。如果测试能够正确执行对应的写操作指令，表明能正确处理缺页异常，则显示

```
“check_pgfault() succeeded!”和“check_vmm() succeeded.”。
```

【实现】缺页异常处理

当启动分页机制以后，如果一条指令或数据的虚拟地址所对应的物理页框不在内存中或者访问的类型有错误（比如写一个只读页或用户态程序访问内核态的数据等），就会发生缺页异常。产生页面异常的原因主要有：

- 目标页面不存在（页表项全为0，即该线性地址与物理地址尚未建立映射或者已经撤销)；
- 相应的物理页面不在内存中（页表项非空，但Present标志位=0，比如在swap分区或磁盘文件上)，这将在下面介绍换页机制实现时进一步讲解如何处理；
- 访问权限不符合(此时页表项P标志=1，比如企图写只读页面).

当出现上面情况之一，那么就会产生页面page fault（#PF）异常。产生异常的线性地址存储在CR2中，并且将 #PF 的类型保存在 error code 中，比如 bit 0 表示是否 PTE_P为0，bit 1 表示是否 write 操作。

产生缺页异常后，CPU硬件和软件都会做一些事情来应对此事。首先缺页异常也是一种异常，所以针对一般异常的硬件处理操作是必须要做的，即CPU在当前内核栈保存当前被打断的程序现场，即依次压入当前被打断程序使用的eflags，cs，eip，errorCode；由于缺页异常的中断号是0xE， CPU把中断0xE服务例程的地址（vectors.S中的标号vector14处）加载到cs和eip寄存器中，开始执行中断服务例程。这时ucore开始处理异常中断，首先需要保存硬件没有保存的寄存器。在vectors.S中的标号vector14处先把中断号压入内核栈，然后再在trapentry.S中的标号__alltraps处把ds、es和其他通用寄存器都压栈。自此，被打断的程序现场被保存在内核栈中。

接下来，在trap.c的trap函数开始了中断服务例程的处理流程，大致调用关系为：

```
trap--> trap_dispatch-->pgfault_handler-->do_pgfault
```

下面需要具体分析一下do_pgfault函数。CPU把引起缺页异常的虚拟地址装到寄存器CR2中，并给出了出错码（tf->tf_err），指示引起缺页异常的存储器访问的类型。而中断服务例程会调用缺页异常处理函数do_pgfault进行具体处理。缺页异常处理是实现按需分页、swap in/out和写时复制的关键之处，后面的小节将分别展开讲述。

ucore中do_pgfault函数是完成缺页异常处理的主要函数，它根据从CPU的控制寄存器CR2中获取的缺页异常的虚拟地址以及根据 error code的错误类型来查找此虚拟地址是否在某个VMA的地址范围内以及是否满足正确的读写权限，如果在此范围内并且权限也正确，这认为这是一次合法访问，但没有建立虚实对应关系。所以需要分配一个空闲的内存页，并修改页表完成虚地址到物理地址的映射，刷新TLB，然后调用iret中断，返回到产生缺页异常的指令处重新执行此指令。如果该虚地址不再某VMA范围内，这认为是一次非法访问。

【注意】

地址空间的管理由虚存管理和页表管理两部分组成。 虚存管理限制了（程序）地址空间的范围以及权限，而页表维护的是实际使用的地址空间以及权限，后者不能比前者有更大的范围或者权限，因为前者是实际管理页表的。比如权限，虚存管理可以规定地址空间的某个范围是可写的，但是页表中却可以标记是read-only的（比如 copy-on-write 的实现），这种冲突可以被内核（通过硬件异常）轻易的捕获到，并进行相应的处理。反过来，如果页表权限比虚存规定的权限更大，内核是没有办法发现这种冲突的。由于虚存管理的存在，内核才能方便的实现更复杂和丰富的操作，比如 share memory、swap 等。 在后续的实验中还会遇到虚存管理只维护用户地址空间（也就是 [USERBASE, USERTOP) 区间)的情况，因为内核地址空间包括虚存和页表都是固定的。

proj8：支持页换入换出

proj8项目组成

编译并运行proj8的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
proj8
|   ├── driver
|   |   ├── …
|   |   ├── ide.c
|   |   └── ide.h
|   ├── fs
|   |   ├── fs.h
|   |   ├── swapfs.c
|   |   └── swapfs.h
|   ├── mm
|   |   ├── ……
|   |   ├── memlayout.h
|   |   ├── pmm.c
|   |   ├── swap.c
|   |   ├── swap.h
|   |   ├── vmm.c
|   |   └── vmm.h
|   ├── sync
|   |   └── sync.h
|   └── trap
|       ├── trap.c
|       └── ……
├── libs
|   ├── hash.c
|   └── ……
├── ……
```

相对于proj7，proj8主要修改和增加的文件如下：

- ide.[ch]：实现了对IDE硬盘的PIO方式的扇区读写功能，用于支持把页换入和换出硬盘。
- swapfs.[ch]：根据页和硬盘扇区的映射关系，实现了在IDE硬盘上的swap文件组织，并实现了把页写入swap文件和从swap文件读入页的功能。需要ide.[ch]的支持。
- swap.[ch]：参考Linux2.4的页替换策略，实现了一个简化的双链表页替换策略。

- memlayout.h：修改Page等关键数据结构，支持双链页替换策略。
- pmm.c：修改page_remove_pte函数，支持双链页替换策略。
- vmm.c：修改do_pgfault函数，支持页的换入换出。
- sync.h：增加lock/unlock支持，支持页的换入换出过程不会出现race condition现象。

proj8编译运行

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc010dfec (phys)
  edata  0xc012faa8 (phys)
  end    0xc0132e20 (phys)
Kernel executable memory footprint: 204KB
memory managment: buddy_pmm_manager
e820map:
  memory: 0009f400, [00000000, 0009f3ff], type = 1.
  memory: 00000c00, [0009f400, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_slab() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
check_pgfault() succeeded!
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
page fault at 0x00000000: K/W [no page found].
page fault at 0x00000000: K/W [no page found].
page fault at 0x00001001: K/W [no page found].
page fault at 0x00001000: K/R [no page found].
page fault at 0x00000000: K/R [no page found].
check_swap() succeeded.
++ setup timer interrupts
100 ticks
```

check_swap函数对ucore在proj8中建立的双链页面置换策略进行了测试，验证了其正确性，下面我们将从原理和实际实现两个方面来分析proj8中实现的页面置换算法。

【原理】页面置换算法

操作系统为何要进行页面置换呢？这是由于操作系统给用户态的应用程序提供了一个虚拟的“大容量”内存空间，而实际的物理内存空间又没有那么大。所以操作系统就就“瞒着”应用程序，只把应用程序中“常用”的数据和代码放在物理内存中，而不常用的数据和代码放在了硬盘这样的存储介质上。如果应用程序访问的是“常用”的数据和代码，那么操作系统已经放置在内存中了，不会出现什么问题。但当应用程序访问它认为应该在内存中的的数据或代码时，如果这些数据或代码不在内存中，则根据上一小节的介绍，会产生缺页异常。这时，操作系统必须能够应对这种缺页异常，即尽快把应用程序当前需要的数据或代码放到内存中来，然后重新执行应用程序产生异常的访存指令。如果在把硬盘中对应的数据或代码调入内存前，操作系统发现物理内存已经没有空闲空间了，这时操作系统必须把它认为“不常用”的页换出到磁盘上去，以腾出内存空闲空间给应用程序所需的数据或代码。

操作系统迟早会碰到没有内存空闲空间而必须要置换出内存中某个“不常用”的页的情况。如何判断内存中哪些是“常用”的页，哪些是“不常用”的页，把“常用”的页保持在内存中，在物理内存空闲空间不够的情况下，把“不常用”的页置换到硬盘上就是页面置换算法着重考虑的问题。容易理解，一个好的页面置换算法会导致缺页异常次数少，也就意味着访问硬盘的次数也少，从而使得应用程序执行的效率就高。

从操作系统原理的角度看，有如下一些页面置换算法：

- 最优 (Optimal) 页面置换算法：由Belady于1966年提出的一种理论上的算法。其所选择的被淘汰页面，将是以后永不使用的或许是在最长的未来时间内不再被访问的页面。采用最佳置换算法，通常可保证获得最低的缺页率。但由于操作系统其实无法预知一个应用程序在执行过程中访问到的若干页中，哪一个页是未来最长时间内不再被访问的，因而该算法是无法实际实现，但可以此算法作为上限来评价其它的页面置换算法。

- 先进先出(First In First Out, FIFO)页面置换算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。只需把一个应用程序在执行过程中已调入内存的页按先后次序链接成一个队列，队列头指向内存中驻留时间最久的页，队列尾指向最近被调入内存的页。这样需要淘汰页时，从队列头很容易查找到需要淘汰的页。FIFO算法只是在应用程序按线性顺序访问地址空间时效果才好，否则效率不高。因为那些常被访问的页，往往在内存中也停留得最久，结果它们因变“老”而不得不被置换出去。FIFO算法的另一个缺点是，它有一种异常现象（Belady现象），即在增加放置页的页帧的情况下，反而使缺页异常次数增多。

- 二次机会（Second Chance）页面置换算法：为了克服FIFO算法的缺点，人们对它进行了改进。此算法在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU中的MMU硬件将把访问位置“1”。当需要找到一个页淘汰时，对于最“老”的那个页面，操作系统去检查它的访问位。如果访问位是0，说明这

个页面老且无用，应该立刻淘汰出局；如果访问位是1，这说明该页面曾经被访问过，因此就再给它一次机会。具体来说，先把访问位位清零，然后把这个页面放到队列的尾端，并修改它的装入时间，就好像它刚刚进入系统一样，然后继续往下搜索。二次机会算法的实质就是寻找一个比较古老的、而且从上一次缺页异常以来尚未被访问的页面。如果所有的页面都被访问过了，它就退化为纯粹的FIFO算法。

- LRU(Least Recently Used，LRU)页面置换算法： FIFO置换算法性能之所以较差，是因为它所依据的条件是各个页调入内存的时间，而页调入的先后顺序并不能反映页是否“常用”的使用情况。最近最久未使用（LRU）置换算法，是根据页调入内存后的使用情况进行决策页是否“常用”。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU置换算法是选择最近最久未使用的页予以淘汰。该算法赋予每个页一个访问字段，用来记录一个页面自上次被访问以来所经历的时间t,，当须淘汰一个页面时，选择现有页面中其t值最大的，即最近最久未使用的页面予以淘汰。

- 时钟（Clock）页面置换算法：也称最近未使用 (Not Used Recently, NUR) 页面置换算法。虽然二次机会算法是一个较合理的算法，但它经常需要在链表中移动页面，这样做既降低了效率，又是不必要的。一个更好的办法是把各个页面组织成环形链表的形式，类似于一个钟的表面。然后把一个指针指向最古老的那个页面，或者说，最先进来的那个页面。时钟算法和第二次机会算法的功能是完全一样的，只是在具体实现上有所不同。时钟算法需要在页表项（PTE）中设置了一位访问位来表示此页表项对应的页当前是否被访问过。当该页被访问时，CPU中的MMU硬件将把访问位置“1”。然后将内存中所有的页都通过指针链接起来并形成一个循环队列。初始时，设置一个当前指针指向某页（比如最古老的那个页面）。操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为“0”，则淘汰该页，把它换出到硬盘上；如果访问位为“1”，这将该页表项的此位置“0”，继续访问下一个页。该算法近似地体现了LRU的思想，且易于实现，开销少。但该算法需要硬件支持来设置访问位，且该算法在本质上与FIFO算法是类似的，惟一不同的是在clock算法中跳过了访问位为1的页。

- 改进的时钟（Enhanced Clock）页面置换算法：在时钟置换算法中，淘汰一个页面时只考虑了页面是否被访问过，但在实际情况中，还应考虑被淘汰的页面是否被修改过。因为淘汰修改过的页面还需要写回硬盘，使得其置换代价大于未修改过的页面。改进的时钟置换算法除了考虑页面的访问情况，还需考虑页面的修改情况。即该算法不但希望淘汰的页面是最近未使用的页，而且还希望被淘汰的页是在主存驻留期间其页面内容未被修改过的。这需要为每一页的对应页表项内容中增加一位引用位和一位修改位。当该页被访问时，CPU中的MMU硬件将把访问位置“1”。当该页被“写”时，CPU中的MMU硬件将把修改位置“1”。这样这两位就存在四种可能的组合情况：（0，0）表示最近未被引用也未被修改，首先选择此页淘汰；（0，1）最近未被使用，但被修改，其次选择；（1，0）最近使用而未修改，再次选择；（1，1）最近使用且修改，最后选择。该算法与时钟算法相比，可进一步减少磁盘的I/O操作次数，但为了查找到一个尽可能适合淘汰的页面，可能需要经过多次扫描，增加了算法本身的执行开销。

【实现】页面置换机制实现（应该放在第四章进程

【实现】页面置换机制实现（应该放在第四章进程管理与调度）

[下面的内容要去掉，并换成局部页置换的实现]

如果要实现页面置换机制，只考虑页面置换算法的设计与实现是远远不够的，还需考虑其他问题：

- 哪些页可以被换出？
- 一个虚拟的页如何与硬盘上的扇区建立对应关系？
- 何时进行换入和换出操作？
- 在proj7的基础上，如何设计数据结构已支持页面置换算法？
- 如何完成页的换入换出操作？

这些问题在下面会逐一进行分析。注意，在proj8中实现了换入换出机制，但现在还没有涉及lab3才实现的用户进程（激活页面置换的用户方）和内核线程（完成页面置换的服务方），所以还无法通过内核线程机制实现一个完整意义上的虚拟内存页面置换功能，并给用户进程提供大于实际物理空间的虚空间。这要等到lab3的proj11才提供上述支持。

可以被换出的页

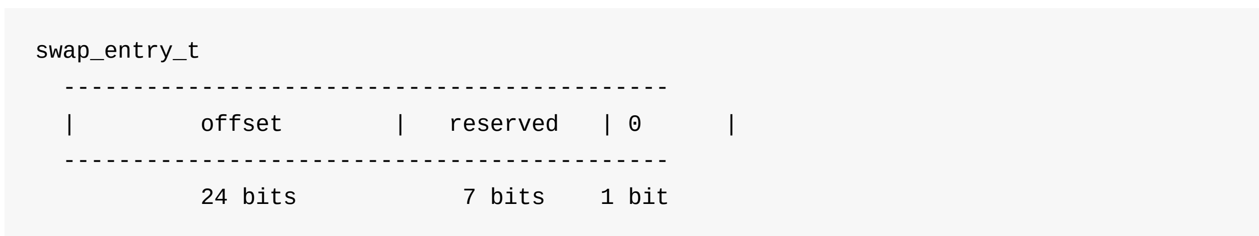
在操作系统的设计中，一个基本的原则是：并非所有的物理页都可以交换出去的，只有映射到用户空间且被用户程序直接访问的页面才能被交换，而被内核直接使用的内核空间的页面不能被换出。这里面的原因是什么呢？操作系统是执行的关键代码，需要保证运行的高效性和实时性，如果在操作系统执行过程中，发生了缺页现象，则操作系统不得不等很长时间（硬盘的访问速度比内存的访问速度慢2~3个数量级），这将导致整个系统运行低效。而且，不难想象，处理缺页过程所用到的内核代码或者数据如果被换出，整个内核都面临崩溃的危险。

但在proj8实现的ucore中，我们只是实现了换入换出机制，还没有设计用户态执行的程序，所以我们在proj8中仅仅通过执行check_swap函数在内核中分配一些页，模拟对这些页的访问，然后直接调用page_launder函数来查询这些页的访问情况并执行页面置换算法换出“不常用”的页到磁盘上。

虚存中的页与硬盘上的扇区之间的映射关系

如果一个页被置换到了硬盘上，那操作系统如何能简捷来表示这种情况呢？在ucore的设计上，充分利用了页表中的PTE来表示这种情况：当一个 PTE 用来描述一般意义上的物理页时，显然它应该维护各种权限和映射关系，以及应该有 PTE_P 标记；但当它用来描述一个被置换出去的物理页时，它被用来维护该物理页与 swap 磁盘上扇区的映射关系，并且该 PTE 不应该由 MMU 将它解释成物理页映射(即没有 PTE_P 标记)，与此同时对应的权限则交由 mm_struct 来维护，当对位于该页的内存地址进行访问的时候，必然导致 #PF，然后内核能够根据 PTE 描述的 swap 项将相应的物理页重新建立起来，并根据虚存所描述的权限重新设置好 PTE 使得内存访问能够继续正常进行。

如果一个页（4KB/页）被置换到了硬盘某8个扇区（0.5KB/扇区），该 PTE的最低位--present位应该为0 （没有 PTE_P 标记），接下来的7位暂时保留，可以用作各种扩展；而原来用来表示页帧号的高24位地址，恰好可以用来表示此页在硬盘上的起始扇区的位置（其从第几个扇区开始）。为了在页表项中区别 0 和 swap 分区的映射，将 swap 分区的一个 page 空出来不用，也就是说一个高24位不为0，而最低位为0的PTE表示了一个放在硬盘上的页的起始扇区号（见swap.h中对swap_entry_t的描述）：



```
swap_entry_t
  -----------------------------------------
  |         offset        |   reserved   | 0      |
  -----------------------------------------
            24 bits            7 bits    1 bit
```

考虑到硬盘的最小访问单位是一个扇区，而一个扇区的大小为512（2^8）字节，所以需要8个连续扇区才能放置一个4KB的页。在ucore中，用了第二个IDE硬盘来保存被换出的扇区，根据proj8的输出信息

```
“ide 1:     262144(sectors), 'QEMU HARDDISK'.”
```

我们可以知道proj8可以保存262144/8=32768个页，即128MB的内存空间。swap 分区的大小是 swapfs_init 里面根据磁盘驱动的接口计算出来的，目前 ucore 里面要求 swap 磁盘至少包含 1000 个 page，并且至多能使用 1\<\<24 个page。

swap.c 中维护了全局的 mem_map 表，用来记录 swap 分区的使用，它会在 swap_init 里面根据 swap 分区的大小进行适当的初始化。表中的每一项是一个 unsigned short 类型的整数，用来记录该 entry 的引用计数。ucore 使用一个非常大的整数 0xFFFF 表示一个 entry （这里 entry 指 swap 分区上连续的 8 个扇区）是空闲的（没有被分配出去），并用 0xFFFE 表示一个 entry 的最大引用计数，通常一个页不会有这么大的引用计数，除非内核崩了。当一个 entry 的引用计数 是 0 的时候，表示该 entry 是可以被回收的，但是我们通常不是真的去回收他，只有当需要分配一个 entry，并且在 mem_map 里面实在找不到空闲的 entry 的时候才会去回收一个引用计数为 0 的 entry。因为一个引用计数为 0 的页很可能上面的数据和对应物理页上的数据一致，这样在换出该页的时候就可以避免写磁盘的代价，而和写磁盘比起来，遍历 mem_map 数组的时间开销总是会小很多。

此外，为了简化实现，swap 只对页表中的数据页进行换出，即只对 PTE 进行操作，而第二级页表是保留不动的。

执行换入换出的时机

check_mm_struct 是在 lab2 里面用来做模块测试时使用的临时的 mm_struct。在 lab3 以后就没有用处了。在proj8中， check_mm_struct变量这个数据结构表示了目前 ucore认为合法的所有虚拟内存空间集合，而mm中的每个vma表示了一段地址连续的合法虚拟空间。当ucore或应用程序访问地址所在的页不在内存时，就会产生 #PF异常，引起调用do_pgfault函数，此函数会判断产生访问异常的地址属于check_mm_struct某个vma表示的合法虚拟地址空间，且保存在硬盘swap文件中（即对应的PTE的高24位不为0，而最低位为0），则是执行页换入的时机，将调用swap_in_page函数完成页面换入。

换出页面的时机相对复杂一些，针对不同的策略有不同的时机。ucore目前大致有两种策略，即积极换出策略和消极换出策略。积极换出策略是指操作系统周期性地（或在系统不忙的时候）主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；消极换出策略是指，只是当试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。

在proj8中，可支持上述两种情况，但都需要到lab3/proj11中才会完整实现。对于第一种积极换出策略，即创建了一个每隔1秒执行一次的内核线程kswapd（在lab3的proj11中第一次出现），实现积极的换出策略。对于第二种消极的换出策略，则是在ucore调用alloc_pages函数获取空闲页时，此函数如果发现无法从页分配器（比如buddy system）获得空闲页，就会进一步调用try_free_pages来唤醒线程kswapd，并将 cpu 让给 kswapd 使得换出某些页，实现一种消极的换出策略。

页面置换算法的数据结构设计

到proj7为止，我们知道目前表示内存中物理页使用情况的变量是基于数据结构Page的全局变量pages数组，pages的每一项表示了计算机系统中一个物理页的使用情况。为了表示物理页可被换出或已被换出的情况，可对Page数据结构进行扩展：

```
struct Page {
    uint32_t flags;   // array of flags that describe the status of the page frame
    swap_entry_t index;      // stores a swapped-out page identifier
list_entry_t swap_link; // swap hash link
……
};
```

首先flag的含义做了扩展：

```
// the page is in the active or inactive page list (and swap hash table)
#define PG_swap   4
// the page is in the active page list
#define PG_active  5
```

前面提到swap.c 里面声明了全局的 mem_map 数据结构，用来存储 swap 分区上的页的使用计数。除此之外，swap.c 里还声明了两个链表，分别是 active_list 和 inactive_list，分别表示已经分配了 swap entry 的且处于“活跃”状态/“不活跃”状态的物理页链表。所有已经分配了 swap entry 的 page 必须处于两个链表中的一个。

当一个物理页 （struct Page） 需要被 swap 出去的时候，首先需要确保它已经分配了一个 swap entry。如果 page数据结构的 flags 设置了PG_swap 为1，则表示该 page 中的 index 是有效的 swap entry的索引值，从而该物理页上的数据可以被写出到 index 所表示的 swap page 上去。

如果一个物理页在硬盘上有一个页备份，则需要记录在硬盘中页备份的位置。Page结构中的index就起到这个记录的作用，它保存了被换出的页的页表项PTE高24位的内容—entry，即硬盘中对应页备份的起始扇区位置值（以字节为单位？？？）。

Page结构中的swap_link保存了以entry为hash索引的链表项，这样根据entry，就可以快速的对 page 数据结构进行查找。但hash数组在哪里呢？

对于在硬盘上有页备份的物理页（简称swap_page），需要统一管理起来，为此在proj8中增加了全局变量：

```
static list_entry_t hash_list[HASH_LIST_SIZE];
```

hash*list*数组就是我们需要的*hash*数组，根据 *index(swap entry)* 索引全部的 *swap* page 的指针，这样通过hash函数

```
#define entry_hashfn(x)                 (hash32(x, HASH_SHIFT))
```

可以快速地根据entry找到对应的page 数据结构。

正如页替换算法描述的那样，把“常用”的可被换出页和“不常用”的可被换出页分别集中管理起来，形成 active_list 和 inactive_list 两个链表。 ucore的页面置换算法会根据相应的准则把它认为“常用”的物理页放到active_list链表中，而把它认为“不常用”的物理页放到inactive_list链表中。一个标记了 PG_swap 的页总是需要在这两个链表之间移动。

前面介绍过 mem_map 数组，他是用来记录 swap_page 的引用次数的。因为 swap 分区上的 page 实际上是某个物理页的数据备份。所以，一个物理页page的 page_ref 与 对应 swap_page 的 mem_map[offset] (offset = swap_offset(entry)) 值的和是这个数据页的真实引用计数。page_ref 表示PTE对该物理页的映射的个数；mem_map 表示 PTE对该 swap 备份

页的 映射个数。当 page_ref 为0 的时候，表示物理页可以被回收；当 mem_map[offset] 为0 的时候，表示 swap page 可以被回收（前面介绍过，可以回收，但是在万不得已的情况下才真的回收）。在后面的实验中还可能涉及到，这里只是简单了解一下。

【注意】

ucore 目前使用的 PIO 的方式读写 IDE 磁盘，这样的好处是，磁盘读入、写出操作可以认为是同步的，即当前CPU需要等待磁盘读写完毕后再进行进一步的工作。由于磁盘操作相对CPU的速度而言是很慢的，这使得会浪费大量的 CPU 时间在等IO操作上。于是我们总是希望能够在 IO 性能上有更大的提升，比如引入 DMA 这种异步的 IO 机制，为了避免后续开发上的各种不便和冲突，我们假设所有的磁盘操作都是异步的（也包括后面的实验），即使目前是通过 PIO 完成的。

假定某page的 flags中的PG_swap 标志位为 1，并且PG_active标志位也为 1，则表示该page 在 swap 的 active_list 中，否则在 inactive_list 中。 active_list 中的页表示活跃的物理页，即页表中可能存在多个 PTE 指向该物理页（这里可以是同一个页表中的多个 entry，在后面lab3的实验里面有了进程以后，也可以是多个进程的页表的多个 entry）；反过来，inactive_list 链表所链接的 page 通常是指没有 PTE 再指向的页。

【注意】

需要强调两点设计因素：

1. 一个 page 是在 active_list 还是在 inactive_list 的条件不是绝对的；
2. 只有 inactive_list 上的页才会被尝试换出。

这两个设计因素的设计起因如下：

1. 我们知道一个 page 换出的代价是很大的（磁盘操作），并且我们假设所有的磁盘操作都是异步的，那么换出一个 active 的页就变得非常不值得。因为在还有多个 PTE 指向他情况下进行换出操作（异步 IO 可能导致进程切换）的较长过程中，这个页可以随时被其它进程写脏。而硬件提供给内核的接口（即页表项PTE的dirty位）使得内核只能知道一个页是否是脏的（不能明确知道一个页的哪个部分是脏的），当这种情况发生时，就导致了一次无效的写出。
2. active_list 和 inactive_list 的维护只能由 与swap有关的集中操作来完成。特别是在lab3/proj11引入 kswapd内核线程之后，所有的内存页换出任务都交给kswapd，这样减少了复杂的同步互斥实现（在lab5中会重点涉及）。
3. 页面换入换出有关的操作需要做的就是尽可能的完成如下三件事情：
    - 将 PG_swap 为 0 的页转变成 PG_swap 为1 的页。即尽可能的给每个物理页分配一个 swap entry（当然前提是有足够大的 swap 分区）。
    - 将页从 active_list 上移动到 inactive_list 上。如果一个页还在 active_list 上，说明还有 PTE 指向此“活跃”的物理页。所以需要在完成内存页换出时断开对这些物理页的引用，把它变成不活跃的（inactive）。只有把所有的PTE对某page的引用都断开

（即 page的page_ref 为0）后，就可以将此page从 active_list 移动到 inactive_list 上。

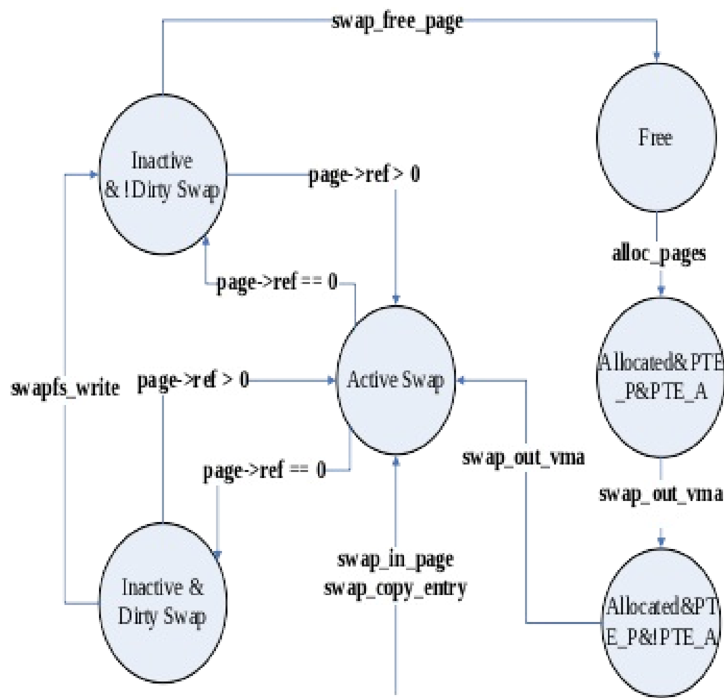
  - 将 inactive_list 上的页写出并释放掉。inactive_list 上的page表示已没有 PTE 指向此page了，那么该page可以被释放，如果该page被写过，那还需把此page换出到swap分区上。如果在整个换出过程（异步 IO）中没有其他进程再写这个物理页（即没有 PTE 在引用它或有PTE引用但页没有写脏），就认为这个物理页是可以安全释放的了。那么将它从 inactive_list 上取下，并调用 page_free函数 实现 page 的回收。

4. 值得注意的是，内存页换出操作只有特定时候才被调用，即通过 执行try_free_pages函数或者定时器机制（在lab3/proj10.4才引入）定期唤醒kswapd内核线程。这样会导致内存页换出操作对两个链表上的数据都不够敏感。比如处于 active_list 上的 page，可能在 kswapd 工作的时候，已经没有 PTE 再引用它了；再如相应的进程退出了，并且相应的地址空间已经被内核回收，从而变成了一个 inactive 的 page；还存在inactive_list 上的page 也可能在换出的时候，其它进程通过 page fault，又将 PTE 指向他，进而变成一个实际上 active 的页。所以说，active 和 inactive 条件并不绝对。

页面置换算法的执行逻辑

其实在proj8中并没有完全实现页面置换算法，只是实现了其中的部分关键函数，并通过check_swap来验证了这些函数的正确性。直到lab3的proj11才形成了完整的页面置换逻辑，而这个页面置换逻辑基本上是改进的时钟算法的一个实际扩展版本。

ucore 采用的页面置换算法是一个全局的页面置换算法，因为它收集了ucore中所有用户态进程（这里可理解为ucore中运行的每个用户态程序）的可换出页，并把这些可换出页中的一部分转换为空闲页。其次它考虑了页的访问情况（根据PTE中PTE_A位的值）和读写情况（根据PTE中PTE_D位的值）。如果页被访问过，则把PTE_A位清零继续找下一页；如果页没有被访问过，这此页就成为了active状态的可换出页，并放入active_list链表中，这时需要把对应的PTE转换成为一个swap entry(高24位保存为硬盘缓存页的起始扇区号，PTE_P位清零)；接着refill_inactive_scan函数会把处于active状态的部分可换出页转换成inactive状态，并放入inactive_list链表中；然后page_launder函数扫描inactive_list中的处于inactive状态的可换出页，如果此页不是dirty的，则把它直接转换成空闲页，如果此页是dirty的，则执行换出操作，把该页换出到硬盘上保存。这个页的状态变化图如下图所示。



ucore的物理页状态变化图

下面具体讲述一下proj11中实现上述置换算法的页面置换逻辑。在proj11中同时实现了积极换出策略和消极换出策略，这都是通过在不同的时机执行kwapd_main函数来完成的。当ucore调用alloc_pages函数以获取空闲页，但物理页内存分配器无法满足请求时，alloc_pages函数将调用tre_free_pages来执行kwapd_main函数（通过直接唤醒线程的方式，在lab3中会进一步讲解），完成对页的换出操作和生成空闲页的操作。这是一种消极换出的策略。另外，ucore设立了每秒执行一次kwapd_main函数（通过设置timer来唤醒线程的方式，在lab3中会进一步讲解），完成对页的换出操作和生成空闲页的操作。这是一种积极换出的策略。

kswapd_main是ucore整个页面置换算法的总控部分，其大致思路是根据当前的空闲页情况查找出足够多的可换出页（swap page），然后根据这些可换出页的访问情况确定哪些是"常用"页，哪些是"不常用"页，最后把"不常用"的页转换成空闲页。思路简单，但具体实现相对复杂。

如前所述，swap 整个流程就是把尽可能多的 page 从变成 PG_active的，并移动到 active list 中；把 active list 中的页尽可能多的变成 inactive list 中页；最后把 inactive list 的页换出（洗净），根据情况处理洗净后的 page 结构（根据 page ref 以及相应的 dirty bit）。所以整个过程的核心是尽可能的断开页表中的 PTE 映射。

当 alloc_pages 执行分配 n 连续物理页失败的时候，则会通过调用tree_free_page来唤醒kswapd线程，此线程执行kswapd_main函数。kswapd 会发现它现在压力很大，需要尽可能的满足分配 n 个连续物理页的需求。既然需求是 n 个连续物理页，那么 kswapd 所需要释放的物理页就应该大于 n 个；每个页可能在某个或者许多个页表的不同的地方有 PTE 映射（特别是 copy on write 之后，这种情况更为普遍），那么 kswapd 所需要断开的 PTE 映射就远远不止 n 个。Linux 实现了能够根据 physical address 在页表中快速定位的数据结构，但是实现起来过于复杂，这里 ucore 采用了一个比较笨的方法，即遍历所有存在的页表结构，断开足

够多的 PTE 映射。这里足够多是个经验公式，采用 n<<5 。当然这也可能失败，那么 kswapd 就会尝试一定次数。当他实在无能为力的时候，也就放弃了。而 alloc_pages 也会不停的调用 try_free_pages 进行尝试，当尝试不停的遭遇失败的时候，程序中会有许多句 warn 来输出这些调试信息。而 Linux 的方案是选择一个占用内存最多的进程杀掉并释放出资源，来尽可能的满足当前程序的需求（注意，这里当前程序是指内核服务或者调用），直到程序从内核态正常退出；ucore 的这种设计显然是过于简单了，不过此是后话。

扫描页表是一项艰巨的任务，因为除了内核空间，用户地址空间有将近 3G 的空间，真正的程序很少能够用这么多。因此，充分利用虚存管理能够很大的提升扫描页表的速度。

接下来，我们需要介绍一下 kswapd_main 是如何一步一步完成 swap 的操作的。正如前面介绍的那样，swap 的任务主要分成三个过程，

现在我们来介绍以下 kswap_main 是如何一步一步完成 swap 的操作的。正如前面介绍过的，swap 需要完成3件事情，下面对应的是这三个操作的具体细节：

1. kswapd_main函数通过循环调用函数swap_out_mm并进一步调用swap_out_vma，来查找ucore中所有存在的虚存空间，并总共断开 m 个 PTE 到物理页的映射（这里需要和进程的概念有所结合，可以理解为每个用户程序都拥有一个自己的虚存空间。但是需要提一句的是，在后的实验中还会遇到虚存空间在多个进程之间的共享；遍历虚存空间而不是遍历每个进程是为了避免某个虚存空间被很多进程共享进而被 kswapd 过度压榨所带来的不公平。可以想象，被过度压榨的虚存空间，同时又由于被很多进程共享从而有很高的概率被使用到，最终必然会导致频繁的 #PF，给系统不必要的负担。还有就是虽然 swap 的任务是断开 m 个 PTE 映射，但是实际上它对每个虚存空间都一次至多提出断开 32 个映射的需求，并循环遍历所有的虚存空间直到 m 得到满足。这样做的目的也是为了保证公平，使得每个虚存空间被交换出去的页的几率是近似相等的。Linux 实际上应该有更好的实现，它根据虚存空间所实际使用的物理页的个数来决定断开的映射的个数。）。这些断开的 PTE 映射所指向的物理页如果没有 PG_active 标记，则需要给他分配一个新的 swap entry，并做好标记，将 page 插入到 active_list 中去（同时也插入到 swap 的哈希表中），然后设置好相应的 page_ref 和 mem_map[offset] 的值，当然，如果找不到空闲的 swap entry 可以分配（比如 swap 分区已经用光了），我们只能跳过这样的 PTE 映射，从下一个地址继续寻找出路；对于原来就已经标记了 PG_swap 的物理页，则只需要完成后面的工作，即调整引用计数就足够了。断开的 PTE 被 swap_entry 取代，并取消 PTE_P 标记，，这样当出现#PF的时候，我们能够直接根据 PTE 上的值得到该页的数据实际是在swap 分区上的哪个位置上。

   现在不必要计较一个 page 究竟是放在 active_list 还是放在 inactive_list 中，更不必要考虑换出这样的操作，这一阶段的工作只是断开 PTE 映射，余下的工作后面会一步步完成。

   当 kswapd 断开足够数量的 PTE 映射以后，这一部分的工作也就完成了。虚存管理中维护了一个 swap_address 的地址，表示上一次 swap 操作结束时的地址，维护这个数据是避免每次 swap 操作都从虚存空间的起始地址开始，从而导致过多数量的重复的无效的遍

历。

当 kswapd 发现自己竭尽所能的遍历都无法满足断开 m 个链接的需求时，该怎么办？我们需要明确的是swap 操作的主要目的是释放物理页，而断开 PTE 映射是一个必要的步骤，作用是尽可能的扩大 inactive_list 中 page 的个数，为物理页的换出提供更大的基数（操作空间），但并不是主要过程。所以为了防止在这一步陷入死循环，kswapd_main 最多会对全部虚存空间的链表尝试 rounds=16 次遍。

2. 通过 page_launder 函数，遍历 inactive_list，实现页的换出。这部分和下面 refill_inactive_list 操作的先后顺序并不那么严格。通俗的解释就是 page_launder 实现的是把 inactive_list 中的 page 洗净，并完成 page 的释放，当然也顺便实现了把实际上活跃的(active) 的 page 从 inactive_list 上取下，放回 active_list 的过程；而 refill_inactive_list 从函数名上可以看出，实际上就是遍历 active_list，把实际上不活跃的(inactive) page 从 active_list 上取下，放到 inactive_list 上，方便下一轮 page_launder 的操作。后者没有什么需要特别强调的，但是 page_launder 是比较复杂的过程，我们需要仔细的分析一下。page_launder 先检查一个 page 的 page_ref 是否 != 0，如果满足，则表示该 page 实际上是 active 的，则把它移动到 active_list 上去；如果不是，则需要对该页进行换出操作，过程如下：注意，下面讨论的是以 page_ref == 0 作为前提的。

   (*) page_launder 的实现，涉及 ucore 内核代码设计的一个重要假设前提，这是第一次涉及，以后的各个模块也会逐步大量涉及。这部分和进程调度又有一定的关联，提前了解一下，有助于理解这部分以后后续其它部分的代码。这个前提就是：ucore 的内核代码是不可抢占的，也就是，执行在内核部分的代码，只要不是以下几种情况，通常可以认为是操作不会被抢占（preemption），即CPU控制权被剥夺。1.主动释放 cpu 执行权限，比如调用 schedule 让其它程序执行；2. 进行同步互斥操作，比如争抢一个信号量、锁；3.进行磁盘等等异步操作，由于 kmalloc 有可能会调用 alloc_pages 来分配页，而 alloc_pages 可能失败进而将cpu 让渡给 kswapd，所以，内核中的 kmalloc 操作可以认为不是一个同步的操作。所有这样的非同步的操作一个可能的问题，就是所有执行比如 kmalloc 这样的函数之前所做出的各种条件判断，在 kmalloc 之后可能都不再成立了。

   你可以理解下面两段程序在运行时的差异，其中list 是一个全局变量，并且可能被任何程序在执行内核服务的时候修改掉：

```
parta:
if (!list_empty(list)) {
    ptr = (uintptr_t)kmalloc(sizeof(uint32_t));
    do_something(list_next(list), buffer);
    kfree(ptr);
}
partb:
ptr = (uintptr_t)kmalloc(sizeof(uint32_t));
if (!list_empty(list)) {
    do_something(list_next(list), buffer);
}
kfree(ptr);
```

除此之外，中断处理的代码也需要进一步考虑进来。当内核尝试修改一部分数据的时候，如果该数据是中断处理流程可能访问的数据，那么内核需要对这次修改屏蔽中断；同理，如果中断处理需可能修改一部分数据，并且内核打算尝试读取该数据，那么内核需要对读操作屏蔽中断，等等。

- 如果一个页的 mem_map 项也为0：前面已经讨论过 mem_map 和 page_ref 之间的关系。如果此时，这个页的 mem_map 项也为0，说明这个时候已经没有 PTE 映射指向它们了，无论是物理页还是 swap 备份页。那么这个页也就没有必要洗净了。可以直接释放物理页以及相应的 swap entry 了。（同时记得处理 swap 有关的链表，以下不再赘述）
- 如果一个页的 mem_map 项不为0，但是没有 PG_dirty 标记：page 数据结构里面有 PG_dirty 标记，swap 部分的代码根据这个标记来判断一个页是否需要被洗净（写到 swap 分区上）。这个 PG_dirty 标记在什么情况下设置，我们稍后会讨论。这种情况可以等价为物理页上的数据和 swap 分区上的数据是一致的，所以不需要洗净该页，因为该物理页本身就已经足够干净了。所以可以安全的和(1) 中的操作一样对该物理页进行释放。
- 如果一个页的确有 PG_dirty 标记：表示该页需要被洗净。ucore 这里的实现存在一个 bug，最新的代码已经修复了这个bug，不过bug对于理解这段代码没有影响 。

  先清楚，洗净一个页，需要调用 swapfs_write 函数，完成将物理页写到磁盘上的操作。前面已经强调过，我们假设所有磁盘操作是异步IO。 先明确一下，当前的状态，page_ref=0 &&mem_map!=0 && PG_dirty，那么写出的过程中就可能发生下面许多种可能的场景：

  - swapfs_write 操作失败了。磁盘操作不像内存操作，它应该允许发生更多的错误。
  - 其它进程又访问到相应的数据页，前面提到过，因为 PTE 已经被修了，所以会产生#PF，内核会根据 PTE 的内容在 swap hash 里面查找到相应的物理页，并将它重新插入到相应的页表中，并更新该 page 的 page_ref 和 mem_map 的

值。整个过程发生时，swapfs_write 还没有结束。那么当完成洗净一个页的操作时（写到swap 分区），swap 部分的代码应该有能力检测出这种变化。也就是在 swapfs_write 之后，需要再判断 page_ref 是否依然满足 inactive 的。

- 和b类似，不过不同的是，这次对物理页进行的是一个写操作。操作完成之后，进程又将该 PTE 指向的页释放掉了。那么当 swapfs_write 返回的时候，它面对的条件，可能就变成了 page_ref=0 &&mem_map=0 &&PG_dirty。它应该能够处理这个变化。
- 和c类似，不同的是，该page有两个不同的 PTE 映射。那么在swapfs_write 操作之前，状态可能是 page_ref =0&&mem_map=2&!PG_dirty，那么当c中的情况发生以后，该物理页的状态就可能变成了 page_ref=0&mem_map=1&PG_dirty了。swap 应该能够处理这种变化。

综上所述，page_launder 部分的代码变得相对复杂很多。大家可以参照程序了解ucore 是怎么解决这种冲突的。 最后，提及一下那个 bug。因为 swapfs_write 是异步操作，并且是对该 page 的操作，ucore 为了保证在操作的过程中，该页不被释放（比如一个进程通过#PF，增加 page_ref到1，然后又通过释放该page减少page_ref到0，进而触发内核执行 page_free 的操作），分别在 swapfs_write 前后获得和释放该 page 的引用（page_ref_inc/page_ref_dec）。但事实证明，这种担心是多余的。理由很简单，当 page_launder 操作一个页的时候，该页是被标记 PG_swap 的，这个标记一方面表示 page 结构中的 index有意义，另一方面也代表了，这样的 page 的释放，只能够由 swap 部分的代码来完成（参见 pmm.c 以及后面 shmem.c 的处理）。所以，swap 在操作该 page 的时候，不可能有程序能够调用free_page释放该 page。

而相反的，mem_map 是一个需要保护的数据。这个是产生 bug 的地方，有兴趣的同学可以去自己理解一下。

此外，可以翻阅一下涉及到 page_ref 修改的 pmm 部分的代码，不难发现，当一个 page 从 PTE 断开的时候，也就是 page_ref 下降的时候，内核会根据 PTE 上的硬件设置的 PTE_D 来设置 PG_dirty。其实这就足够了。因为 PG_dirty 并不需要时时刻刻都十分的准确，只要在 swap 尝试判断该 page 是否需要洗净的时候，PG_dirty 是正确的，就足够了。所以只需要保证每次 page_ref 下降的时候，PG_dirty 是正确的即可。除此之外，在对每个页分配 swap_entry 的时候，需要保证标记 PG_dirty，因为毕竟是刚刚分配的，物理页的数据还从来没有写出去过。

总结一下，页换入换出的实现很复杂，但是相对独立。并且正是由于 ucore 的内核代码不可抢占使得实现变得相对容易一些。只要是不涉及 IO 操作，大部分过程都可以认为处于不可抢占的内核执行过程。

proj9.1：实现共享内存

实现共享内存功能的目的是为将来（lab5中才需要）不同进程（process）之间能够通过共享内存实现数据共享。共享内存机制其实是一种进程间通信的手段。proj9/proj9.1完成了不同页表中（目前仅局限于父子进程之间,lab3才涉及）的虚拟地址共享同一块物理地址空间的功能。由于目前的实现仅限于有亲属关系的进程，这实际上意味着这些具有共享物理地址空间的虚拟地址空间也是相同的。

相关数据结构

这部分的具体实现工作主要在kern/mm/shmem.[ch]和其他一些文件中。根据前面的分析，我们知道在mm_struct 层面上管理了基于同一个页表的vma集合，这些集合表示了当前可“合法”使用（即使没有对应的物理内存）的所有虚拟地址空间集合。当前的vma_struct定义如下：

```
struct vma_struct {
    struct mm_struct *vm_mm;
    uint32_t vm_start;
    uint32_t vm_end;
    uint32_t vm_flags;
    vma_entry_t vma_link;
};
```

这 个是proj9.1以前的 vma_struct 结构定义。由于vma中并没有描述此虚拟地址对应的物理地址空间，所以无法确保不同页表中描述同一虚拟地址空间的vma指向同一个物理地址空间，所以不同页表间的内存无法实现共享。于是我们可以对vma_struct增加下面两个域（field）：

```
struct shmem_struct *shmem;
size_t shmem_off;
```

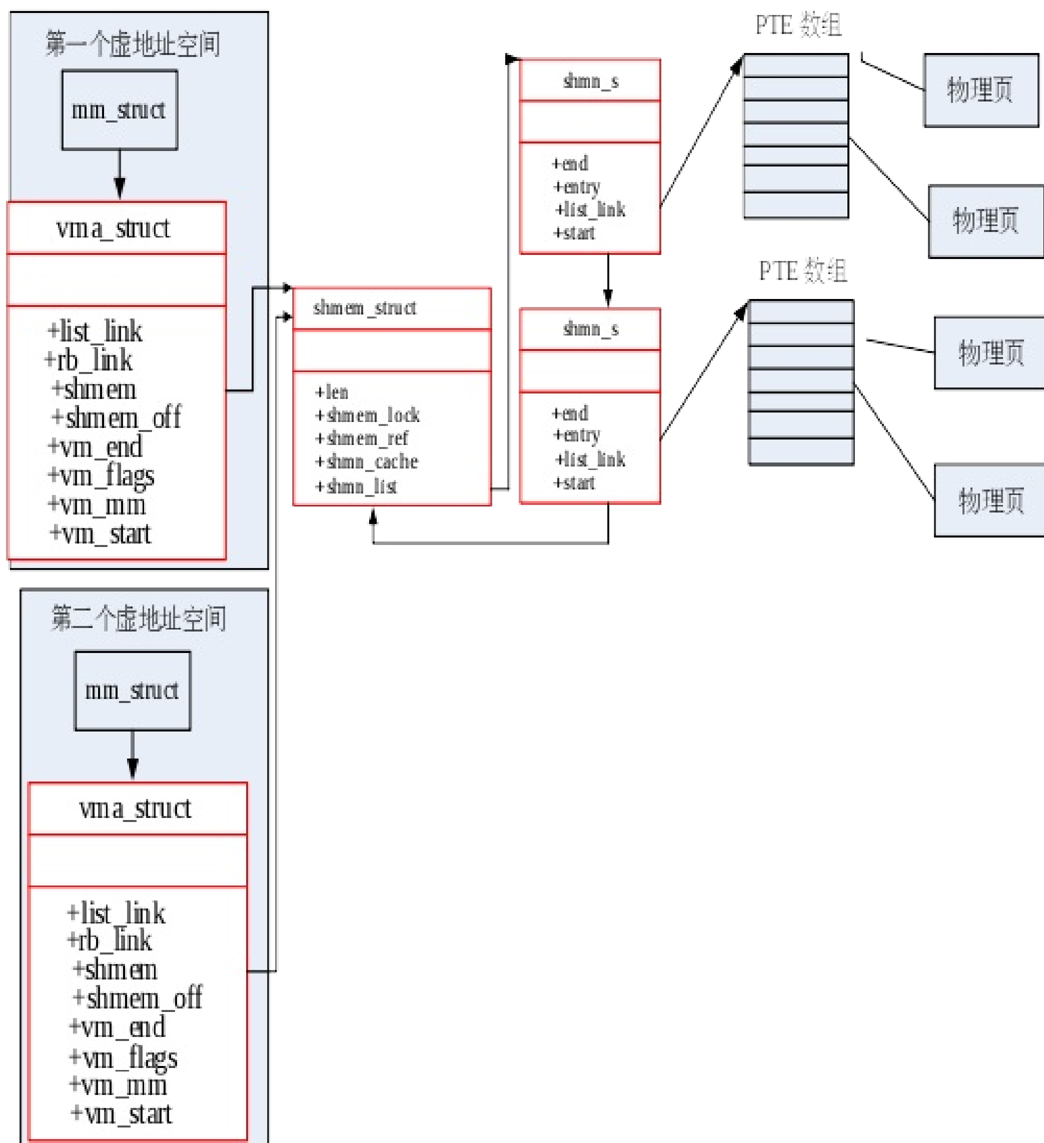
shmem的作用是统一维护不同mm_struct结构（即不同页表）中描述具有共享属性的同一虚拟地址空间的唯一物理地址空间。如果 vma->flags 里面有 VM_SHARE，就表示 vma->shmem 有意义，他指向一个 shmem_struct结构的指针。shmem_struct结构定义如下：

```
struct shmem_struct {
    list_entry_t shmn_list;
    shmn_t *shmn_cache;
    size_t len;
    atomic_t shmem_ref;
    lock_t shmem_lock;
};
```

shmem_struct包含了list_entry_t结构的shmn_list，此链表的元素是shmn_t结构的共享页描述（包含某共享虚拟页的信息（page 或者 swap entry），为了维护起来简便，里面借用了页表的PTE描述方式，除了 PTE_P 用来区分是否是一个物理页以外，没有任何其它权限标记，所以后面提到的 PTE 应该是带引号的），所以此链表就是用来存储虚拟地址空间的PTE集合，即此共享的虚拟地址空间对应的唯一的物理地址空间的映射关系。shmem_ref指出了当前有多少进程共享此共享虚拟空间。shmn_t是用来描述一段共享虚拟空间的信息，可理解为一个shmem node结构，其定义如下：

```
typedef struct shmn_s {
    uintptr_t start;
    uintptr_t end;
    pte_t *entry;
    list_entry_t list_link;
} shmn_t;
```

在这个结构中，entry保存了一块（4KB）连续的虚拟空间的PTE数组，可以理解为一个二级页表，这个页表最大可以描述4MB的连续虚拟空间对应的物理空间地址信息。entry 中的每一项用于保存 physical address | PTE_P 或者 swap_entry。这样能最大限度节约内存，并且能很快通过entry项计算出 对应的struct page。 而list_link是用来把自身连接属于同一shmem_struct结构的域-shmn_list链表，便于shmem_struct结构的变量对共享空间进行管理。这样就可以形成如下图所示的共享内存布局：



创建和访问共享内存的实现逻辑

为了创建一块共享内存，首先需要在内核中调用do_shmem函数(此函数要到lab3的proj12才会出现)，do_shmem函数并进一步调用shmem_create函数创建一个shmem_struct结构的变量，然后再调用mm_map_shmem函数来创建一个vma结构的变量，并设置vma的属性为VM_SHARE，并把vma->shmem指向shmem结构变量。这样就初步建立好了一个共享内存虚拟空间。

由于并没有给这块共享内存虚拟空间实际分配物理空间，所以在第一次访问此vma中某地址时，会产生缺页异（page fault），并触发do_pgfault函数被调用。此函数如果发现是“合法”的共享内存虚拟空间出现的地址访问错时，将调用shmem的处理函数shmem_get_entry，来查找此共享内存虚拟子空间所在的其他虚拟地址空间中（具有不同的页表）是否此虚地址对应的PTE项存在，如果不存在，则表明这是第一次访问这个共享内存区域，就可以创建一

个物理页，并在shmn_s的entry中记录虚拟页与物理页的映射关系；如果存在，则在本虚拟地址空间中建立此虚地址对应的PTE项，这样就可以确保不同页表中的共享虚拟地址空间共享一个唯一的物理地址空间。

shmem结构中增加一个计数器，在执行复制某虚拟地址空间的copy_mm函数的时，如果vm_flags 有 VM_SHARE，则仅增加 shmem 计数器shmem_ref，而不用再创建一个shmem变量；同理，释放 具有共享内存属性的vma 时，应该减少shmem计数器shmem_ref，当shmem计数器shmem_ref减少到0的时候，应该释放 shmem 所占有的资源。

另外，在shmem 里面不能记录地址，因为不同 vma 可能map到不同的地址上去，因此它只维护一个 page 的大小。上面提到的 shmem_off 的作用是定位页面。具有共享属性的 vma 创建的时候，shmem_off = 0。当 vma->vm_start 增加的时候（只可能变大，因为内核不支持它减小，unmap 的时候可能导致 vma 前面部分的 unmap，这就可能会让 vm_start 变大），应该将 vm_start 的增量赋给 shmem_off，以保证剩下的 shmem 能够访问正确的位置。这样在访问共享内存地址addr发生缺页异常的时候，此地址对应的页在 shmem_struct 里面的 PTE 数组项的索引index 应该等于 (addr - vma->vm_start + vma->shmem_off) / PGSIZE

【注意】

在页换出操作中， 尝试换出一页的条件是页的 page_ref 到 0。为了防止 share memory 的 page 被意外的释放掉，shmem 结构也会增加相应数据页的引用计数。那么对于一个share memory 的数据页，是不是就不能换出了？前面提到，页换出操作的第一步是扫描所有的虚拟地址空间，那么页换出操作就完全有能力知道当前扫描的 vma 是普通的 vma 还是对应的 share memory 的vma。正如 swap.c 里面看到的那样，swap 断开一个 page 以后，如果发现当前 vma 是share memory 的，并且 page_ref 是1，那么可以确定的是这个最后一个 page_ref 是在 shmem 结构中。那么swap 也同时将该 share memory 上的PTE 断开，就满足了 page_launder 的换出条件。

share memory 上的 entry 换成了 swap entry 带来的坏处也很明显，因为标记share 的 vma 如果一开始没有页表内容，需要通过 #PF 从shmem 里面得到相应的 PTE。但是不幸的是，得到的是 swap entry，那么只能再通过第二次 #PF，才能将swap entry 替换成数据页。

proj9.2：实现写时复制

proj9.2实现了写时复制（Copy On Write，简称COW）的主要功能，为lab3高效地创建子进程打下了基础。COW有何作用？这里又不得不提前讲讲lab3中的子进程创建。不同的进程应该具有不同的物理内存空间，当用户态进程发出fork( )系统调用来创建子进程时，ucore可复制当前进程（父进程）的整个地址空间，这样就有两块不同的物理地址空间了，新复制的那一块物理地址空间分配给子进程。这种行为是非常耗时和占内存资源的，因为它需要为子进程的页表分配页面，复制父进程的每一个物理内存页。如果子进程加载一个新的程序开始执行(这个过程会释放掉原来申请的全部内存和资源)，这样前面的复制工作就白做了，完全没有必要。

为了解决上述问题，ucore采用一种有效的COW机制。其设计思想相对简单：父进程和子进程之间共享（share）页面而不是复制（copy）页面。但只要页面被共享，它们就不能被修改，即是只读的。注意此共享是指父子进程共享一个表示内存空间的mm_struct结构的变量。当父进程或子进程试图写一个共享的页面，就产生一个页访问异常，这时内核就把这个页复制到一个新的页面中并标记为可写。注意，原来的页面仍然是写保护的。当其它进程试图写入时，ucore检查写进程是否是这个页面的唯一属主（通过判断page_ref 和 swap_page_count 即 mem_map 中相关 entry 保存的值的和是否为1。注意区分与share memory的差别，share memory 通过 vma 中的 shmem 实现，这样的 page 是直接标记为共享的，而不是 copy on write，所以也没有任何冲突）；如果是，它把这个页面标记为对这个进程是可写的。

在具体实现上，ucore调用dup_mmap函数，并进一步调用copy_range函数来具体完成对页表内容的复制，这样两个页表表示同一个虚拟地址空间（包括对应的物理地址空间），且还需修改两个页表中每一个页对应的页表项属性为只读，但。在这种情况下，两个进程有两个页表，但这两个页表只映射了一块只读的物理内存。同理，对于换出的页，也采用同样的办法来共享一个换出页。综上所述，我们可以总结出：如果一个页的PTE属性是只读的，但此页所属的VMA描述指出其虚地址空间是可写的，则这样的页是COW页。

当对这样的地址空间进行写操作的时候，会触发do_pgfault函数被调用。此函数如果发现是COW页，就会调用alloc_page函数新分配一个物理页，并调用memcpy函数把旧页的内容复制到新页中，并最后调用page_insert函数给当前产生缺页错的进程建立虚拟页地址到新物理页地址的映射关系（即改写PTE，并设置此页为可读写）。

这里还有一个特殊情况，如果产生访问异常的页已经被换出到硬盘上了，则需要把此页通过swap_in_page函数换入到内存中来，如果进一步发现换入的页是一个COW页，则把其属性设置为只读，然后异常处理结束返回。但这样重新执行产生异常的写操作，又会触发一次内存访问异常，则又要执行上一段描述的过程了。

Page结构的ref域用于跟踪共享相应页面的进程数目。只要进程释放一个页面或者在它上面执行写时复制，它的ref域就递减；只有当ref变为0时，这个页面才被释放。

进程管理与调度

“进程”（process）是20世纪60年代初首先由MIT的MULTICS系统和IBM公司的CTSS/360系统率先引入的概念。简单地说，进程是一个正在运行的程序。但传统的程序本身是一组指令的集合，是一个静态的概念。程序在一方面无法描述程序在内存中的动态执行情况，即无法从程序的字面上看出它何时执行，何时结束；另一方面，在内存中可存在多个程序，这些程序分时复用一个CPU，但无法清楚地表达程序间关系（比如父子关系、同步互斥关系等）。因此，程序这个静态概念已不能如实反映多程序并发执行过程的特征。

为了从根本上描述程序动态执行过程的性质，计算机科学家引入了“进程（Process）”概念。在计算机系统中，由于CPU的速度非常快（现在的通用CPU主频达到2GHz是很平常的事情），只让它做一件事情无法充分发挥其能力。我们可以“同时”运行多个程序，这个“同时”，其实是操作系统给用户造成的一个“错觉”。大家知道，CPU是计算机系统中的硬件资源。为了提高CPU的利用率，在内存中的多个程序可分时复用CPU，即如果一个程序因某个事件而不能继续执行时，就可把CPU占用权转交给另一个可运行程序。为了刻画多各程序的并发执行的过程，就引入了“进程”的概念。从操作系统原理上看，一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。操作系统中的进程管理需要协调多道程序之间的关系，解决对处理器分配调度策略、分配实施和回收等问题，从而使得处理器资源得到最充分的利用。

操作系统需要管理这些进程，使得这些进程能够公平、合理、高效地分时使用CPU，这需要操作系统完成如下主要任务：

- 进程生命周期管理：创建进程、让进程占用CPU执行、让进程放弃CPU执行、销毁进程；
- 进程分派（dispatch）与调度（scheduling）：设定进程占用/放弃CPU的时机、根据某种策略和算法选择将占用的CPU（这就是调度），完成进程切换；
- 进程内存空间保护：给每个进程一个受到保护的地址空间，确保进程有独立的地址空间，不会被其他进程非法访问；
- 进程内存空间等资源共享：提供内存等资源共享机制，可以使得不同进程可共享内存等资源；
- 系统调用机制：给用户进程提供访问操作系统功能的接口，即系统调用接口。

本章内容主要涉及操作系统的进程管理与调度，并能够利用lab2的虚存管理功能实现高效的进程中的内存管理。读者通过阅读本章的内容并动手实践相关的lab3和lab4种的9个project实验：

- Proj10：创建进程控制块和内核线程。
- Proj10.1：实现用户进程、读和加载ELF格式执行程序、一个简单的调度器，以及提供创建（fork）/execve（执行）/ 放弃对CPU的占用（yield）等系统调用实现。

- Proj10.2：完成等待子进程结束（wait），杀死进程（kill），进程自己退出（exit）等系统调用，从而完善了进程的生命周期管理。
- Proj10.3：实现sys_brk系统调用和相应的用户进程内存管理，从而与lab2的内存管理进一步联合在一起。
- Proj10.4：让进程可以睡眠和被唤醒。
- Proj11：创建kswapd内核线程来专门处理内存页的换入和换出。
- Proj12：基于进程间内存共享（proj9.1）实现父子进程数据共享，并实现了用户态的线程机制。
- Proj13：设计实现了通用的调度器框架
- Proj13.1/Proj13.2：在通用调度器框架下实现了轮转（RoundRobin， RR）调度器/多级反馈队列（Multi Level Feedback Queue， MLFQ）调度器

可以掌握如下知识：

- 与操作系统原理相关
- 进程管理：进程状态和进程状态转换
- 进程管理：进程创建、进程删除、进程阻塞、进程唤醒
- 进程管理：父子进程的关系和区别
- 进程管理：进程中的内存管理
- 进程管理：用户进程、内核进程、用户线程、内核线程的关系区别
- 进程管理：线程的特征和实现机制
- 进程调度：进程调度算法
- 操作系统原理之外
- 页面换入换出的内核线程实现技术
- 父子进程数据共享实现
- 通用的调度器框架
- 进程切换的实现细节

本章内容中主要涉及进程管理的重要功能主要有两个：

- 进程生命周期的管理：如何高效地创建进程、切换进程、删除进程和管理进程对资源的需求（内存和CPU）涉及一系列的动态管理机制。线程的加入使得整个系统的执行效率更高，这需要根据线程的特点设计与进程不同的线程组织和管理机制。
- 进程调度算法：进程调度（部分教科书也称为处理器调度）算法主要是选择响应时间决定应该由哪个进程占用CPU来执行。这里需要确保通过调度来提高整个系统的吞吐量和减少响应时间。

为了让读者能够从实践上来理解进程管理和调度的基本原理，我们设计了上述实验，主要的功能逐步实现如下所示：

- 首先需要能够对运行的程序进行管理，这需要一个“档案”，进程控制块（Process Control Block）；
- 有了进程控制块，我们就可以实现不同特点的进程或线程，这里首先实现了相对简单的

内核线程；

- 为了能够执行应用程序，还需通过进程控制块实现用户进程的管理，能够创建/删除/阻塞/唤醒进程，从而能够对用户进程的整个生命周期进行全程管理；
- 由于在内存中有多个进程，但只有一个CPU，所以需要设计合理的调度器，让不同进程能够分时复用CPU，从而提高整个系统的吞吐量和减少响应时间。

创建并执行内核线程

实验目标

ucore在lab2完成了内存管理。一个程序如果要加载到内存中运行，通过ucore的内存管理就可以分配合适的空间了。接下来就需要考虑如何使用CPU来“并发”执行多个程序。

操作系统把一个程序加载到内存中运行，这个运行的程序会经历从“出生”到“死亡”的整个“生命”历程。这个运行程序的整个执行过程就是进程。为了记录、描述和管理程序执行的动态变化过程，需要有一个数据结构，这个就是进程控制块。一个进程与一个进程控制块一一对应。为此，ucore就需要建立合适的进程控制块数据结构，并基于进程控制块来完成对进程的管理。

proj10概述

实现描述

project10是lab3的第一个项目，它基于lab2的最后一个项目proj9.2。主要就是扩展了进程控制块的数据结构，并基于进程控制块实现了对进程的初步管理。并通过创建两个内核线程idleproc和init_main来实现了对CPU的分时使用。

项目组成

```
proj10
├─  ……
|   ├── process
|   |   ├── entry.S
|   |   ├── proc.c
|   |   ├── proc.h
|   |   └── switch.S
|   ├── schedule
|   |   ├── sched.c
|   |   └── sched.h
|   ├── sync
|   |   └── sync.h
|   └── trap
|       ├── trap.c
|       ├── ……
|       └── trapentry.S
├── libs
|   ├── ……
|   └── unistd.h
└─  ……
14 directories, 77 files
```

相对与proj9.2，proj10增加了7个文件，修改了相对重要的3个文件。主要修改和增加的文件如下：

- process/proc.[ch]：实现了进程控制块的定义和基于进程控制块的各种进程管理函数，实现了对进程生命周期管理的绝大部分功能。
- process/entry.S：内核线程的起始入口处和结束处理（通过调用do_exit完成实际结束工作）。
- process/switch.S：实现了进程上下文（context）切换的函数switch_to，由于与硬件相关，所以直接采用汇编实现。
- schedule/sched.[ch]：实现了一个先进先出（First In First Out）策略的进程调度。
- trap/trap.c,trapentry.S：
- unistd.h：定义了一系列系统调用号，为后续用户进程访问内核功能做准备，这里暂时用不上。

编译运行

编译并运行proj10的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
thuos:~/oscourse/ucore/i386/lab3_process/proj10$ make qemu
(THU.CST) os is loading ...

Special kernel symbols:
entry  0xc010002c (phys)
etext  0xc0113bd7 (phys)
edata  0xc013fab0 (phys)
end    0xc0144e74 (phys)
Kernel executable memory footprint: 276KB
……
check_mm_shm_swap: step2, dup_mmap ok.
check_mm_shm_swap() succeeded.
++ setup timer interrupts
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
kernel panic at kern/process/proc.c:317:
process exit!!.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
从上图可以看到，proj10创建了一个内核线程init_main，然后启动内核线程initproc运行，此线程调用init_main函数完成了其主要的工作，即输出了一些信息：
this initproc, pid = 1, name = "init"
To U: "Hello world!!".
To U: "en.., Bye, Bye. :)"
然后就“死亡”了，所占用的资源被回收。由于现在没有其他值得执行的线程，所以ucore就进入到kernel debug monitor了。到底是在哪里判断并进入monitor的呢？我们只需在“K>”提示符后面输入backtrace，就可得到如下输出：
K> backtrace
ebp:0xc7eb0ee8 eip:0xc0101c86 args:0x00000000 0x00000000 0xc7eb0f68 0xc0102489
    kern/debug/kdebug.c:298: print_stackframe+21
ebp:0xc7eb0ef8 eip:0xc010258b args:0x00000000 0xc7eb0f1c 0x00000000 0x00000000
    kern/debug/monitor.c:147: mon_backtrace+10
ebp:0xc7eb0f68 eip:0xc0102489 args:0xc013fac0 0x00000000 0xc011784a 0xc7eb0fdc
    kern/debug/monitor.c:93: runcmd+134
ebp:0xc7eb0f98 eip:0xc010250d args:0x00000000 0xc7eb0fdc 0x0000013d 0xc7eb0fcc
    kern/debug/monitor.c:114: monitor+91
ebp:0xc7eb0fc8 eip:0xc0102b2f args:0xc0117825 0x0000013d 0xc0117839 0xc0144e44
    kern/debug/panic.c:30: __panic+106
ebp:0xc7eb0fe8 eip:0xc0112bc7 args:0x00000000 0xc01178b8 0x00000000 0x00000010
    kern/process/proc.c:317: do_exit+33
K>
```

这里可以清楚的看到，在proc.c的317行（do_exit函数内）调用了panic函数，导致进入了monitor。idleproc 和 initproc 是 ucore 里面两个特殊的内核线程，它们是不允许退出的，所以 do_exit 里面直接调用 panic 了，对于其它普通的进程而言，do_exit 实际上还有很多工作要做，后续章节会进一步讲到。上述执行过程其实包含了对进程整个生命周期的管理。下面我们将从原理和实现两个方面对此进行进一步阐述。

【原理】进程的属性与特征解析

为了让多个程序能够使用CPU执行任务，我们需要设计进程控制块，需要进一步管理进程。但到底如何设计进程控制块，如何管理进程？如果我们对进程的属性和特征了解不够，则无法有效地设计进程控制块和实现进程管理。

再一次回到进程的定义：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。这里有四个关键词：程序、数据集合、执行、动态执行过程。从CPU的角度来看，所谓程序就是一段特定的指令机器码序列而已。CPU会一条一条地取出在内存中程序的指令并按照指令的含义执行各种功能。所谓数据集合就是使用的内存，所谓执行就是让CPU工作。那么这个数据集合和执行其实体现了进程对资源的占用。动态执行过程体现了程序执行的不同“生命”阶段：诞生、工作、休息/等待、死亡。如果这一段指令执行完毕，也就意味着进程结束了。从开始执行到执行结束是一个进程的全过程。那么操作系统需要管理进程的什么？如果计算机系统中只有一个进程，那操作系统的工作就简单了。其实就是管理进程执行的指令，进程占用的资源，进程执行的状态。这可归结为对一个进程内的管理工作。但实际上在计算机系统的内存中，可以放很多程序，这也就意味着操作系统需要管理多个进程，那么，还需要做有关进程间的其他管理工作包括：进程调度、进程间的数据共享、进程间执行的同步互斥关系（lab5涉及）等。下面逐一进行解析。

指令执行安全管理

CPU的指令有一般指令和特权指令之分，那么由不包含特权指令的一般指令集合组成了用户程序，而由特权指令加上一般指令形成的核心软件就是操作系统了。CPU在执行时可处于不同的特权模式：用户态模式和核心态模式，在用户态模式只能执行一般指令，在核心态模式除了可以执行一般指令外，还可以执行特权指令。如果这里的程序是指的一般应用程序或用户程序，不是操作系统，那么我们把在CPU处于用户态特权模式下的应用程序的执行过程称为用户进程。如果这里的程序指的是操作系统，那么我们把在CPU处于核心态特权模式下的操作系统的执行过程称为内核进程。回想lab1和lab2中内容，物理内存在操作系统的内存管理下也有类型，可以分为用户态内存和核心态内存，这两种内存形成了CPU可能访问的所有内存空间。内核进程可以干任何事情，所以对于各种内存，各种指令，它都能访问和执行。用户进程和内核进程对内存和指令的权限如下所示：

	CPU 特权模式	本身的用户态内存	其他进程的用户态内存	核心态内存	一般指令	特权指令
用户进程	用户态模式	可访问	不可访问	不可访问	可执行	执行无效
内核进程	核心态模式	可访问	可访问	可访问	可执行	可执行

用户进程本身要完成的工作细节由一条一条指令具体体现。对于一般指令完成“合法”的功能，操作系统不用管理。操作系统要管理的是用户进程的“非法”指令。简单地说，“非法”指令包括：特权指令、引起异常的一般指令（比如除零操作）和访问不属于它的内存空间。

而且操作系统会在用户进程执行前，设置一个用户进程执行的用户态环境，即如果程序的指令流一开始执行，操作系统就让CPU处于用户态特权模式。而且还要分配一定的物理空间，建立页表，给进程一个用户态虚拟内存空间，这样一个用户态进程就可以在这个限定的虚拟内存空间中正常执行一般指令了。一旦用户进程执行了“非法”指令，则CPU会产生异常，CPU使用权将转到操作系统中，从而让操作系统能够对执行“非法”指令的用户进程进行管理，比如让进程退出、给进程分配更大的内存空间等。这样操作系统能够确保用户进程在执行过程中无法破坏在内存中的其他用户进程和操作系统本身。

资源管理

在计算机系统中，进程会占用内存和CPU，这都是有限的资源，如果不进行合理的管理，资源会耗尽或无法高效公平地使用，从而会导致计算机系统中的多个进程执行效率很低，甚至由于资源不够而无法正常执行。

对于用户态进程而言，操作系统是它的“上帝”，操作系统给了用户态进程可以运行所需的资源，最基本的资源就是内存和CPU。在lab2中涉及的内存管理方法和机制可直接应用到进程的内存资源管理中来。在有多个进程存在的情况下，对于CPU这种资源，则需要通过进程调度来合理选择一个进程，并进一步通过进程分派和进程切换让不同的进程分时复用CPU，执行各自的工作。进程调度的核心是各种进程调度算法，而调度算法的评价指标是高效、合理、公平、系统吞吐量大、响应时间短等。另外，对于无法剥夺的共享资源，如果资源管理不当，多个进程会出现死锁或饥饿现象

状态管理

用户进程有不同的状态（也可理解为“生命”的不同阶段），当操作系统把程序的放到内存中后，这个进程就“诞生”了，不过还没有开始执行，但已经消耗了内存资源，处于“创建”状态；当进程准备好各种资源，就等能够使用CPU时，进程处于“就绪”状态；当进程终于占用CPU，程序的指令被CPU一条一条执行的时候，这个进程就进入了“工作”状态，也称“运行”状态，这时除了进一步占用内存资源外，还占用了CPU资源；当这个进程等待某个资源而无法继续执行时，进程可放弃CPU使用，即释放CPU资源，进入“等待”状态；当程序指令执行完毕，进程进入了“死亡”状态。

这些状态的转换时机需要操作系统管理起来，而且进程的创建和清除等工作必须由操作系统提供，而且从“运行”态与“就绪”态/“等待”态之间的转换，涉及到保存和恢复进程的“执行现场”，也成为进程上下文，这是确保进程即使“断断续续”地执行，也能正确完成工作的必要保证。

系统调用

操作系统把用户进程现在在用户态特权模式下执行，这使得用户进程无法完成各种重要的工作，比如获取内存、访问硬盘内容等。为了解决这个问题，用户进程可通过执行系统调用来通知操作系统帮助它来完成这些需要在特权态下才能执行的重要工作。执行系统调用会使得CPU从用户态特权模式切换到核心态特权模式，在用户进程的用户态执行现场（用户态进程运行上下文）也需要保存在操作系统中，当操作系统完成用户进程请求的工作后，还需根据保存的用户态执行现场恢复用户进程的正常执行。

进程与线程

一个进程拥有一个存放程序和数据的的虚拟地址空间以及其他资源。一个进程基于程序的指令流执行，其执行过程可能与其它进程的执行过程交替进行。因此，一个具有执行状态（运行态、就绪态等）的进程是一个被操作系统调度并分派的单位。在大多数操作系统中，这两个特点是进程的主要本质特征。但这两个特征相对独立，操作系统可以把这两个特征分别进行管理。

这样可以把拥有资源所有权的单位通常仍称作进程，对资源的管理成为进程管理；把指令执行流的单位称为线程，对线程的管理就是线程调度和线程分派。对属于同一进程的所有线程而言，这些线程共享进程的虚拟地址空间和其他资源，但每个线程都有一个独立的栈，还有独立的线程运行上下文用于包含表示线程执行现场的寄存器值等信息。

在多线程环境中，进程被定义成资源分配与保护的单位，与进程相关联的信息主要有存放进程映像的虚拟地址空间等。在一个进程中，可能有一个或多个线程，每个线程有线程执行状态（运行、就绪、死亡等），保存上次运行时的线程上下文、线程的执行栈等。考虑到CPU有不同的特权模式，参照进程的分类，线程又可进一步细化为用户线程和内核线程。

到目前为止，我们就可以明确用户进程、核心进程、用户线程、核心线程的区别了。从本质上看，线程就是一个特殊的不用拥有资源的轻量级进程，在ucore的调度和执行管理中，并没有区分线程和进程。且由于ucore内核中的所有内核进程共享一个内核地址空间和其他资源，所以这些内核进程都是内核线程。理解了进程或线程的上述属性和特征，我们就可以进行进程/线程管理的设计与实现了。但是为了叙述上的简便，在分析proj12（proj12实现了用户线程）以前，以下用户态的进程/线程统称为用户进程，而内核进程/线程则统称为内核线程。

【实现】设计进程控制块

在proj10中，进程管理信息用struct proc_struct表示，在kern/process/proc.h中定义如下：

```
struct proc_struct {
    enum proc_state state;        // Process state
    int pid;                        // Process ID
    int runs;                       // the running times of Proces
    uintptr_t kstack;             // Process kernel stack
    volatile bool need_resched; // need to be rescheduled to release CPU?
    struct proc_struct *parent; // the parent process
    struct mm_struct *mm;         // Process's memory management field
    struct context context;     // Switch here to run process
    struct trapframe *tf;       // Trap frame for current interrupt
    uintptr_t cr3;                // the base addr of Page Directroy Table(PDT)
    uint32_t flags;               // Process flag
    char name[PROC_NAME_LEN + 1];  // Process name
    list_entry_t list_link;    // Process link list
    list_entry_t hash_link;    // Process hash list
};
```

在上述域中，与进程管理各个相关层面的对应关系如下表所示：

安全管理	资源管理	状态管理	系统调用	进程/线程
内存安全：mm	内存资源:mm 页表首地址:cr3 内核堆栈: kstack 统计占用 CPU 的次数：runs	进程状态:state "运行"状态的执行现场:context	产生中断或执行系统调用时执行现场:tf	内核线程：mm=NULL 进程间关系：parent 进程 id:pid 进程名字:name

下面重点解释一下几个比较重要的域：

- mm ：内存管理的信息，包括内存映射列表、页表指针等。mm里有个很重要的项pgdir，记录的是该进程使用的一级页表的物理地址。
- state：进程所处的状态。
- parent ：用户进程的父进程（创建它的进程）。在所有进程中，只有一个进程没有父进程，就是内核创建的第一个内核线程idleproc。内核根据这个父子关系建立进程的树形结构，用于维护一些特殊的操作，例如确定哪些进程是否可以对另外一些进程进行什么样的操作等等。
- context：进程的上下文，用于进程切换（参见switch.S）。在 ucore 中，所有的进程在内核中也是相对独立的（例如独立的内核堆栈以及上下文等等）。使用 context 保存寄存器的目的就在于在内核态中能够进行上下文之间的切换。实际利用context进行上下文切换的函数是switch_to，在kern/process/switch.S中定义。

- tf：中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。除此之外，ucore 内核允许嵌套中断。因此为了保证嵌套中断发生时tf 总是能够指向当前的 trapframe，ucore 在内核栈上维护了 tf 的链，可以参考 trap.c::trap函数 做进一步的了解。
- cr3: cr3 保存页表的物理地址，目的就是进程切换的时候方便直接使用 lcr3 实现页表切换，避免每次都根据 mm 来计算 cr3。mm 数据结构是用来实现用户空间的虚存管理的，但是内核线程没有用户空间，它执行的只是内核中的一小段代码（通常是一小段函数），所以它没有 mm 结构，也就是NULL。当某个进程是一个普通用户态进程的时候，PCB 中的 cr3 就是 mm 中页表（pgdir）的物理地址；而当它是内核线程的时候，cr3 等于 boot_cr3。 而boot_cr3指向了ucore启动时建立好的饿内核虚拟空间的页目录表首地址。
- kstack: 每个进程都有一个内核栈，并且位于内核地址空间的不同位置。对于内核线程，该栈就是运行时的程序使用的栈；而对于普通进程，该栈是发生特权级改变的时候使保存被打断的硬件信息用的栈。Ucore在创建进程时分配了 2 个连续的物理页（参见 memlayout.h）作为内核栈的空间。这个栈很小，所以内核中的代码应该尽可能的紧凑，并且避免在栈上分配大的数据结构，以免栈溢出，导致系统崩溃。kstack记录了分配给该进程/线程的内核栈的位置。主要作用有以下几点。首先，当内核准备从一个进程切换到另一个的时候，需要根据 kstack 的值正确的设置好 tss （可以回顾一下在lab1中讲述的 tss 在中断处理过程中的作用），以便在进程切换以后再发生中断时能够使用正确的栈。其次，内核栈位于内核地址空间，并且是不共享的（每个进程/线程都拥有自己的内核栈），因此不受到 mm 的管理，当进程退出的时候，内核能够根据 kstack 的值快速定位栈的位置并进行回收。ucore 的这种内核栈的设计借鉴的是 linux 的方法（但由于内存管理实现的差异，它实现的远不如 linux 的灵活），它使得每个进程/线程的内核栈在不同的位置，这样从某种程度上方便调试，但同时也使得内核对栈溢出变得十分不敏感，因为一旦发生溢出，它极可能污染内核中其它的数据使得内核崩溃。如果能够通过页表，将所有进程的内核栈映射到固定的地址上去，能够避免这种问题，但又会使得进程切换过程中对栈的修改变得相当繁琐。感兴趣的同学可以参考 linux kernel 的代码对此进行尝试。

为了管理系统中所有的进程控制块，ucore维护了如下全局变量（位于 kern/process/proc.c）：

- static struct proc *current; //当前占用CPU，处于“运行”状态进程控制块指针。通常这个变量是只读的，只有在进程切换的时候才进行修改，并且整个切换和修改过程需要保证操作的原子性，目前至少需要屏蔽中断，可以参考 switch_to 的实现，后面也会介绍到。linux 的实现很有意思，它将进程控制块放在进程内核栈的底部，这使得任何时候 current 都可以根据内核栈的位置计算出来的，而不用维护一个全局变量。这样使得一致性的维护以及多核的实现变得十分的简单和高效。感兴趣的同学可以参考 linux kernel 的代码。
- static struct proc *initproc; //指向第一个用户态进程（proj10以后）
- static list_entry_t hash_list[HASH_LIST_SIZE]; //所有进程控制块的哈希表，这样

proc_struct中的域hash_link将基于pid链接入这个哈希表中。

- list_entry_t proc_list;// 所有进程控制块的双向线性列表，这样proc_struct中的域list_link将链接入这个链表中。

【实现】创建并执行内核线程

既然建立了进程控制块，我们就可以通过进程控制块来创建具体的进程了。首先，我们考虑最简单的内核线程，它通常只是内核中的一小段代码或者函数，没有用户空间。而由于在操作系统启动后，已经对整个核心内存空间进行了管理，通过设置页表建立了核心虚拟空间（即boot_cr3指向的二级页表描述的空间）。所以内核中的所有线程都不需要再建立各自的页表，只需共享这核心虚拟空间就可以访问整个物理内存了。

创建第0个内核线程idleproc

在init.c::kern_init函数调用了proc.c::proc_init函数。proc_init函数启动了创建内核线程的步骤。 首先当前的执行上下文（从kern_init 启动至今）就可以看成是一个内核线程的上下文，为此 ucore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化而将其打造成第0个内核线程 -- idleproc。具体步骤如下：

首先调用alloc_proc函数来通过kmalloc函数获得proc_struct结构的一块内存—proc，这就是第0个进程控制块了，并把proc进行初步初始化（即把proc_struct中的各个域清零）。但有些域设置了特殊的值：

```
 proc->state = PROC_UNINIT; //设置进程为“初始”态
 proc->pid = -1;              //进程的pid还没设置好
 proc->cr3 = boot_cr3;       //进程在内核中使用的内核页表的起始地址
```

上述三条语句中,第一条设置了进程的状态为“初始”态，这表示进程已经 “出生”了，正在获取资源茁壮成长中；第二条语句设置了进程的pid为-1，这表示进程的“身份证号”还没有办好；第三条语句表明进程如果在内核运行，则采用为内核建立的页表，即设置了在内核的页表的起始地址，这也可进一步看出所有进程的内核虚地址空间（也包括物理地址空间）是相同的。既然内核进程共用一个映射内核空间的页表，这表示所有这些内核空间对所有内核进程都是“可见”的，所以更精确地说，这些内核进程都应该是内核线程。

接下来，proc_init函数对idleproc内核线程进行进一步初始化：

```
idleproc->pid = 0;
idleproc->state = PROC_RUNNABLE;
idleproc->kstack = (uintptr_t)bootstack;
idleproc->need_resched = 1;
set_proc_name(idleproc, "idle");
```

需要注意前4条语句。第一条语句给了idleproc合法的身份证号--0，这名正言顺地表明了idleproc是第0个内核线程。“0”是第一个的表示方法是计算机领域所特有的，比如C语言定义的第一个数组元素的小标也是“0”。第二条语句改变了idleproc的状态，使得它从“出生”转到了“准备工作”，就差ucore调度它执行了。第三条语句设置了idleproc所使用的内核栈的起始地址。需要注意以后的其他进程/线程的内核栈都需要通过分配获得，因为ucore启动时设置的内核栈直接分配给idleproc使用了。第四条很重要，因为ucore希望当前CPU应该做更有用的工作，而不是运行idleproc这个“无所事事”的内核线程，所以把idleproc->need_resched设置为“1”，结合idleproc的执行主体--cpu_idle函数的实现，可以清楚看出如果当前idleproc在执行，则只要此标志为1，马上就调用schedule函数要求调度器切换其他进程执行。

【问题】为何说**idleproc**的执行主体是**cpu_idle**函数？

创建第1个内核线程initproc

第0个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行cpu_idle函数开始过退休生活了。但接下来还需创建其他进程来完成各种工作，但idleproc自己不想做，于是就通过调用kernel_thread函数创建了一个内核线程init_main。在proj10中，这个子内核线程的工作就是输出一些字符串，然后就返回了（参看init_main函数）。但在后续的proj中，init_main的工作就是创建特定的其他内核线程或用户进程。下面我们来分析一下创建内核线程的函数kernel_thread：

```
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf_struct;
    memset(&tf_struct, 0, sizeof(struct trapframe));
    tf_struct.tf_cs = KERNEL_CS;
    tf_struct.tf_ds = tf_struct.tf_es = tf_struct.tf_ss = KERNEL_DS;
    tf_struct.tf_regs.reg_ebx = (uint32_t)fn;
    tf_struct.tf_regs.reg_edx = (uint32_t)arg;
    tf_struct.tf_eip = (uint32_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf_struct);
}
```

注意，kernel_thread函数采用了局部变量tf来放置保存内核线程的临时中断帧，并把中断帧的指针传递给do_fork函数，而do_fork函数会调用copy_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间。

给中断帧分配完空间后，就需要构造新进程的中断帧，具体过程是：首先给tf进行清零初始化，并设置中断帧的代码段（tf.tf_cs）和数据段(tf.tf_ds/tf_es/tf_ss)为内核空间的段（KERNEL_CS/ KERNEL_DS），这实际上也说明了initproc内核线程在内核空间中执行。而initproc内核线程从哪里开始执行呢？tf.tf_eip的指出了是kernel_thread_entry（位于kern/process/entry.S中），kernel_thread_entry是entry.S中实现的汇编函数，它做的事情很简单：

```
kernel_thread_entry:        # void kernel_thread(void)

    pushl %edx              # push arg
    call *%ebx              # call fn

    pushl %eax              # save the return value of fn(arg)
    call do_exit            # call do_exit to terminate current thread
```

从上可以看出，kernel_thread_entry函数主要为内核线程的主体fn函数做了一个准备开始和结束运行的“壳”，及把函数fn的参数arg（保存在edx寄存器中）压栈，然后调用fn函数，把函数返回值eax寄存器内容压栈，调用do_exit函数退出线程执行。

do_fork是创建线程的主要函数。kernel_thread函数通过调用do_fork函数最终完成了内核线程的创建工作。下面我们来分析一下do_fork函数的实现。do_fork函数主要做了以下6件事情：

1．分配并初始化进程控制块（alloc_proc函数）；

2．分配并初始化内核栈（setup_stack函数）；

3．根据clone_flag标志复制或共享进程内存管理结构（copy_mm换生）；

4．设置进程在内核（将来也包括用户态）正常运行和调度所需的中断帧和执行上下文（copy_thread函数）；

5．把设置好的进程控制块放入hash_list和proc_list两个全局进程链表中；

6．自此，进程已经准备好执行了，把进程状态设置为“就绪”态。

这里需要注意的是，如果上述前3步执行没有成功，则需要做对应的出错处理，把相关已经占有的内存释放掉。copy_mm函数目前只是把current->mm设置为NULL，这是由于目前proj10只能创建内核线程，proc->mm描述的是进程用户态空间的情况，所以目前mm还用不上。copy_thread函数做的事情比较多，代码如下：

```
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t)forkret;
    proc->context.esp = (uintptr_t)(proc->tf);
}
```

此函数首先在内核堆栈的顶部设置中断帧大小的一块栈空间，并在此空间中拷贝在kernel_thread函数建立的临时中断帧的初始值，并进一步设置中断帧中的栈指针esp和标志寄存器eflags，特别是eflags设置了FL_IF标志，这表示此内核线程在执行过程中，能响应中断，打断当前的执行。执行到这步后，此进程的中断帧就建立好了，对于initproc而言，它的中断帧如下所示：

```
//所在地址位置
initproc->tf= (proc->kstack+KSTACKSIZE) – sizeof (struct trapframe);
//具体内容
initproc->tf.tf_cs = KERNEL_CS;
initproc->tf.tf_ds = initproc->tf.tf_es = initproc->tf.tf_ss = KERNEL_DS;
initproc->tf.tf_regs.reg_ebx = (uint32_t)init_main;
initproc->tf.tf_regs.reg_edx = (uint32_t) ADDRESS of "Hello world!!";
initproc->tf.tf_eip = (uint32_t)kernel_thread_entry;
initproc->tf.tf_regs.reg_eax = 0;
initproc->tf.tf_esp = esp;
initproc->tf.tf_eflags |= FL_IF;
```

设置好中断帧后，最后就是设置initproc的执行现场（也称进程上下文, process context）了。只有设置好执行现场后，一旦ucore调度器选择了initproc执行，就需要根据initproc->context中保存的执行现场来恢复initproc的执行这里设置了initproc的执行现场中主要的两个信息：上次停止执行时的下一条指令地址context.eip和上次停止执行时的堆栈地址context.esp。其实initproc还没有执行过，所以这其实就是initproc实际执行的第一条指令地址和堆栈指针。可以看出，由于initproc的中断帧占用了实际给initproc分配的栈空间的顶部，所以initproc就只能把栈顶指针context.esp设置在initproc的中断帧的起始位置。根据context.eip的赋值，可以知道initproc实际开始执行的地方在forkret函数处。至此，initproc内核线程已经做好准备执行了。

调度并执行内核线程initproc

在ucore执行完proc_init函数后，就创建好了两个内核线程：idleproc和initproc，这时ucore当前的执行现场就是idleproc，等到执行到init函数的最后一个函数cpu_idle之前，ucore的所有初始化工作就结束了，idleproc将通过执行cpu_idle函数让出CPU，给其它内核线程执行，具体过程如下：

```
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
     ……
```

首先，判断当前内核线程idleproc的need_resched是否不为0，回顾本节中“创建第一个内核线程idleproc”中的描述，proc_init函数在初始化idleproc中，就把idleproc->need_resched置为1了，所以会马上调用schedule函数找其他处于“就绪”态的进程执行。

ucore在proc10中只实现了一个最简单的FIFO调度器，其核心就是schedule函数。它的执行逻辑很简单：

1．设置当前内核线程current->need_resched为0；

2．在proc_list队列中查找下一个处于“就绪”态的线程或进程next；

3．找到这样的进程后，就调用proc_run函数，保存当前进程current的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换。

至此，新的进程next就开始执行了。由于在proc10中只有两个内核线程，且idleproc要让出CPU给initproc执行，我们可以看到schedule函数通过查找proc_list进程队列，只能找到一个处于“就绪”态的initproc内核线程。并通过proc_run和进一步的switch_to函数完成两个执行现场的切换，具体流程如下：

1．让current指向next内核线程initproc；

2．设置任务状态段ts的特权态0下的栈顶指针esp0为next内核线程initproc的内核栈的栈顶，即next->kstack + KSTACKSIZE ；

3．设置CR3寄存器的值为next内核线程initproc的页目录表起始地址next->cr3，这实际上是完成进程间的页表切换；

4．由switch_to函数完成具体的两个线程的执行现场切换，即切换各个寄存器，当switch_to函数执行完“ret”指令后，就切换到initproc执行了。

这里需要注意，在第二步设置任务状态段ts的特权态0下的栈顶指针esp0的目的是建立好内核线程或将来用户线程在执行特权态切换（从特权态0<-->特权态3，或从特权态3<-->特权态3）时能够正确定位处于特权态0时进程的内核栈的栈顶，而这个栈顶其实放了一个trapframe结构的内存空间。如果是在特权态3发生了中断/异常/系统调用，则CPU会从特权态3-->特权态0，且CPU从此栈顶开始压栈来保存被中断/异常/系统调用打断的用户态执行现场；如果是在特权态0发生了中断/异常/系统调用，则CPU会从特权态还是0，且 CPU从当前栈指针esp所指的位置开始压栈保存被中断/异常/系统调用打断的内核态执行现场。反之，当执行完对中断/异常/系统调用打断的处理后，最后会执行一个“iret”指令。在执行此指令之前，CPU的当前栈指针esp一定指向上次产生中断/异常/系统调用时CPU保存的被打断的指令地址CS和EIP，“iret”指令会根据ESP所指的保存的址CS和EIP恢复到上次被打断的地方继续执行。

在页表设置方面，由于idleproc和initproc都是共用一个内核页表boot_cr3，所以此时第三步其实没用，但考虑到以后的进程有各自的页表，其起始地址各不相同，只有完成页表切换，才能确保新的进程能够正常执行。

第四步proc_run函数调用switch_to函数，参数是前一个进程和后一个进程的执行现场context。在上一节“设计进程控制块”中，描述了context结构包含的要保存和恢复的寄存器。我们在看看switch.S中的switch_to函数的执行流程：

```
.globl switch_to
switch_to:                      # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax          # eax points to from
    popl 0(%eax)                # save eip !popl
    movl %esp, 4(%eax)
    ……
    movl %ebp, 28(%eax)
    # restore to's registers
    movl 4(%esp), %eax          # not 8(%esp): popped return address already
                                # eax now points to to
    movl 28(%eax), %ebp
    ……
    movl 4(%eax), %esp
    pushl 0(%eax)               # push eip
    ret
```

首先，保存前一个进程的执行现场，前两条汇编指令（如下所示）保存了进程在返回switch_to函数后的指令地址到context.eip中

```
movl 4(%esp), %eax          # eax points to from
popl 0(%eax)                # save eip !popl
```

在接下来的7条汇编指令完成了保存前一个进程的其他7个寄存器到context中的相应域中。至此前一个进程的执行现场保存完毕。接下来就是恢复向一个进程的执行现场，这其实就是上述保存过程的逆执行过程，即从context的高地址的域ebp开始，逐一把相关域的值赋值给对应的寄存器，倒数第二条汇编指令“pushl 0(%eax)”其实把context中保存的下一个进程要执行的指令地址context.eip放到了堆栈顶，这样接下来执行最后一条指令“ret”时，会把栈顶的内容赋值给EIP寄存器，这样就切换到下一个进程执行了，即当前进程已经是下一个进程了。

再回到proj10中，ucore会执行进程切换，让initproc执行。在对initproc进行初始化时，设置了initproc->context.eip = (uintptr_t)forkret，这样，当执行switch_to函数并返回后，initproc将执行其实际上的执行入口地址forkret。而forkret会调用位于kern/trap/trapentry.S中的forkrets函数执行，具体代码如下：

```
.globl __trapret
__trapret:
    # restore registers from stack
    popal

    # restore %ds and %es
    popl %es
    popl %ds

    # get rid of the trap number and error code
    addl $0x8, %esp
    iret

.globl forkrets
forkrets:
    # set stack to this new process's trapframe
    movl 4(%esp), %esp       //把esp指向当前进程的中断帧
    jmp __trapret
```

可以看出，forkrets函数首先把esp指向当前进程的中断帧，从_trapret开始执行到iret前，esp指向了current->tf.tf_eip，而如果此时执行的是initproc，则current->tf.tf_eip= kernel_thread_entry，initproc->tf.tf_cs = KERNEL_CS，所以当执行完iret后，就开始在内核中执行kernel_thread_entry函数了，而initproc->tf.tf_regs.reg_ebx = init_main，所以在kernl_thread_entry中执行“call %ebx”后，就开始执行initproc的主体了。Initprocde的主体函数很简单就是输出一段字符串，然后就返回到kernel_tread_entry函数，并进一步调用do_exit执行退出操作了。本来do_exit应该完成一些资源回收工作等，但这些不是proj10涉及的，而是由后续的proj来完成。 至此，proj10的主要工作描述完毕

创建并执行用户进程

实验目标

到proj10为止，ucore还一直在核心态“打转”，没有到用户态执行。其实这也是对操作系统的要求，操作系统就要呆在核心态，才能管理整个计算机系统。但应用程序员也需要编写各种应用软件，且要在计算机系统上运行。如果把这些应用软件都作为内核线程来执行，那系统的安全性就无法得到保证了。操作系统的观点是：操作系统程序员编写的操作系统模块是可信的、高效的，对计算机的物理资源了如指掌，可在计算机的核心态执行；但应用程序员编写的应用软件是不可信的，不必了解计算机的物理资源，且需要操作系统提供服务，故放在用户态执行，无法轻易破坏操作系统和其他用户态的进程通过系统调用获得操作系统的服务。所以，ucore要提供用户态进程的创建和执行机制，给应用程序执行提供一个用户态运行环境。

proj10.1概述

实现描述

proj10.1是lab3的第二个project。它在proj10的基础上实现了对用户态进程的支持，主要扩展设计了用户进程执行的用户地址空间、对用户进程访存错误的异常处理、提供用户进程执行效率的按需分页和写时复制的支持、加载并执行依附在ucore内核文件的用户执行程序、实现系统调用机制等。

项目组成

```
proj10.1
├── ……
├── kern
│   ├── ……
│   ├── mm
│   │   ├── memlayout.h
│   │   ├── vmm.c
│   │   └── vmm.h
│   ├── process
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── ……
│   ├── syscall
│   │   ├── syscall.c
│   │   └── syscall.h
│   └── trap
│       ├── trap.c
│       └── ……
└── user
    ├── badsegment.c
    ├── divzero.c
    ├── faultread.c
    ├── faultreadkernel.c
    ├── hello.c
    ├── libs
    │   ├── initcode.S
    │   ├── panic.c
    │   ├── stdio.c
    │   ├── syscall.c
    │   ├── syscall.h
    │   ├── ulib.c
    │   ├── ulib.h
    │   └── umain.c
    ├── pgdir.c
    ├── softint.c
    ├── testbss.c
    └── yield.c

17 directories, 97 files
```

相对于proj10，proj10.1主要增加了有关系统调用实现的syscall.[ch]和测试ucore对用户进程支持的各种用户态程序和库文件，并对相关的内核文件进行了修改。主要修改和增加的文件如下：

- mm/memlayout.h：定义了用户态空间的范围，具体可看“Virtual memory map”的ASCII图注释。
- mm/vmm.[ch]：在vmm.h文件中，扩展了mm_struct数据结构，支持多进程对mm_struct的引用计数和互斥访问，和针对mm_struct结构的引用计数操作和互斥操作；在vmm.c中，主要增加了部分函数防止多进程（比如父子进程、多线程等）同时访问进程的mm进

程内存管理数据结构。
- process/proc.[ch]：对一系列进程管理相关函数进行了扩展，并新实现了部分函数。这是内核改动最大的部分。
- syscall/syscall.[ch]：新增加的部分，提供用户态进程所需的系统服务的操作系统层接口，根据系统调用号，在转到具体的服务功能函数中完成用户态进程的服务请求。
- trap/trap.c：增加对系统调用的初始化和处理，扩展对访存错误异常的处理。
- user/*：实现应用程序所需的基本库函数支持，提供实现系统调用的用户层接口。

编译运行

编译并运行proj10.1的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc0114ba3 (phys)
  edata  0xc018656f (phys)
  end    0xc018b934 (phys)
Kernel executable memory footprint: 559KB
memory management: buddy_pmm_manager
……
check_mm_shm_swap() succeeded.
++ setup timer interrupts
kernel_execve: pid = 1, name = "hello".
Hello world!!.
I am process 1.
hello pass.
kernel panic at kern/process/proc.c:379:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

上述执行输出相对于proj10没有太多变化，只是出现了“kernel_execve: … hello pass”等字符串。但其实在其背后，涉及创建用户态进程，把执行代码加载到用户态线程地址空间，执行系统调用等一系列操作。为了更好地理解proj10.1的设计和实现方案，我们先需要在了解一下用户态进程的特征。下面我们将从原理和实现两个方面对此进行进一步阐述。

【原理】用户进程的特征

从内核线程到用户进程

在proj10中设计实现了进程控制块，并实现了内核线程的创建和简单的调度执行。但proj10中没有在用户态执行用户进程的管理机制，既无法体现用户进程的地址空间，以及用户进程间地址空间隔离的保护机制，不支持进程执行过程的用户态和核心态之间的切换，且没有用户进程的完整状态变化的生命周期。其实没有实现的原因是内核线程不需要这些功能。那内核线程相对于用户态线程有何特点呢？

但其实我们已经在proj10中看到了内核线程，内核线程的管理实现相对是简单的，其特点是直接使用操作系统（比如ucore）在初始化中建立的内核虚拟内存地址空间，不同的内核线程之间可以通过调度器实现线程间的切换，达到分时使用CPU的目的。由于内核虚拟内存空间是一一映射计算机系统的物理空间的，这使得可用空间的大小不会超过物理空间大小，所以操作系统程序员编写内核线程时，需要考虑到有限的地址空间，需要保证各个内核线程在执行过程中不会破坏操作系统的正常运行。这样在实现内核线程管理时，不必考虑涉及与进程相关的虚拟内存管理中的缺页处理、按需分页、写时复制、页换入换出等功能。如果在内核线程执行过程中出现了访存错误异常或内存不够的情况，就认为操作系统出现错误了，操作系统将直接宕机。在ucore中，就是调用panic函数，进入内核调试监控器kernel_debug_monitor。

内核线程管理思想相对简单，但编写内核线程对程序员的要求很高。从理论上讲（理想情况），如果程序员都是能够编写操作系统级别的"高手"，能够勤俭和高效地使用计算机系统中的资源，且这些"高手"都为他人着想，具有奉献精神，在别的应用需要计算机资源的时候，能够从大局出发，从整个系统的执行效率出发，让出自己占用的资源，那这些"高手"编写出来的程序直接作为内核线程运行即可，也就没有用户进程存在的必要了。

但现实与理论的差距是巨大的，能编写操作系统的程序员是极少数的，与当前的应用程序员相比，估计大约差了3~4个数量级。如果还要求编写操作系统的程序员考虑其他未知程序员的未知需求，那这样的程序员估计可以成为是编程界的"上帝"了。

从应用程序编写和运行的角度看，既然程序员都不是"上帝"，操作系统程序员就需要给应用程序员编写的程序提供一个既"宽松"又"严格"的执行环境，让对内存大小和CPU使用时间等资源的限制没有仔细考虑的应用程序都能在操作系统中正常运行，且即使程序太可靠，也只能破坏自己，而不能破坏其他运行程序和整个系统。"严格"就是安全性保证，即应用程序执行不会破坏在内存中存在的其他应用程序和操作系统的内存空间等独占的资源；"宽松"就算是方便性支持，即提供给应用程序尽量丰富的服务功能和一个远大于物理内存空间的虚拟地址空间，使得应用程序在执行过程中不必考虑很多繁琐的细节（比如如何初始化PCI总线和外设等，如果管理物理内存等）。

让用户进程正常运行的用户环境

在操作系统原理的介绍中，一般提到进程的概念其实主要是指用户进程。从操作系统的设计和实现的角度看，其实用户进程是指一个应用程序在操作系统提供的一个用户环境中的一次执行过程。这里的重点是用户环境。用户环境有啥功能？用户环境指的是什么？

从功能上看，操作系统提供的这个用户环境有两方面的特点。一方面与存储空间相关，即限制用户进程可以访问的物理地址空间，且让各个用户进程之间的物理内存空间访问不重叠，这样可以保证不同用户进程之间不能相互破坏各自的内存空间，利用虚拟内存的功能（页换入换出）。给用户进程提供了远大于实际物理内存空间的虚拟内存空间。

另一方面与执行指令相关，即限制用户进程可执行的指令，不能让用户进程执行特权指令（比如修改页表起始地址），从而保证用户进程无法破坏系统。但如果不能执行特权指令，则很多功能（比如访问磁盘等）无法实现，所以需要提供某种机制，让操作系统完成需要特权指令才能做的各种服务功能，给用户进程一个“服务窗口”,用户进程可以通过这个“窗口”向操作系统提出服务请求，由操作系统来帮助用户进程完成需要特权指令才能做的各种服务。另外，还要有一个“中断窗口”，让用户进程不主动放弃使用CPU时，操作系统能够通过这个“中断窗口”强制让用户进程放弃使用CPU，从而让其他用户进程有机会执行。

基于功能分析，我们就可以把这个用户环境定义为如下组成部分：

- 建立用户虚拟空间的页表和支持页换入换出机制的用户内存访存错误异常服务例程：提供地址隔离和超过物理空间大小的虚存空间。
- 应用程序执行的用户态CPU特权级：在用户态CPU特权级，应用程序只能执行一般指令，如果特权指令，结果不是无效就是产生“执行非法指令”异常；
- 系统调用机制：给用户进程提供“服务窗口”；
- 中断响应机制：给用户进程设置“中断窗口”，这样产生中断后，当前执行的用户进程将被强制打断，CPU控制权将被操作系统的中断服务例程使用。

用户态进程的执行过程分析

在这个环境下运行的进程就是用户进程。那如果用户进程由于某种原因下面进入内核态后，那在内核态执行的是什么呢？还是用户进程吗？首先分析一下用户进程这样会进入内核态呢？回顾一下lab1，就可以知道当产生外设中断、CPU执行异常（比如访存错误）、陷入（系统调用），用户进程就会切换到内核中的操作系统中来。表面上看，到内核态后，操作系统取得了CPU控制权，所以现在执行的应该是操作系统代码，由于此时CPU处于核心态特权级，所以操作系统的执行过程就就应该是内核进程了。这样理解忽略了操作系统的具体实现。如果考虑操作系统的具体实现，应该如果来理解进程呢？

从进程控制块的角度看，如果执行了进程执行现场（上下文）的切换，就认为到另外一个进程执行了，及进程的分界点设定在执行进程切换的前后。到底切换了什么呢？其实只是切换了进程的页表和相关硬件寄存器，这些信息都保存在进程控制块中的相关域中。所以，我们

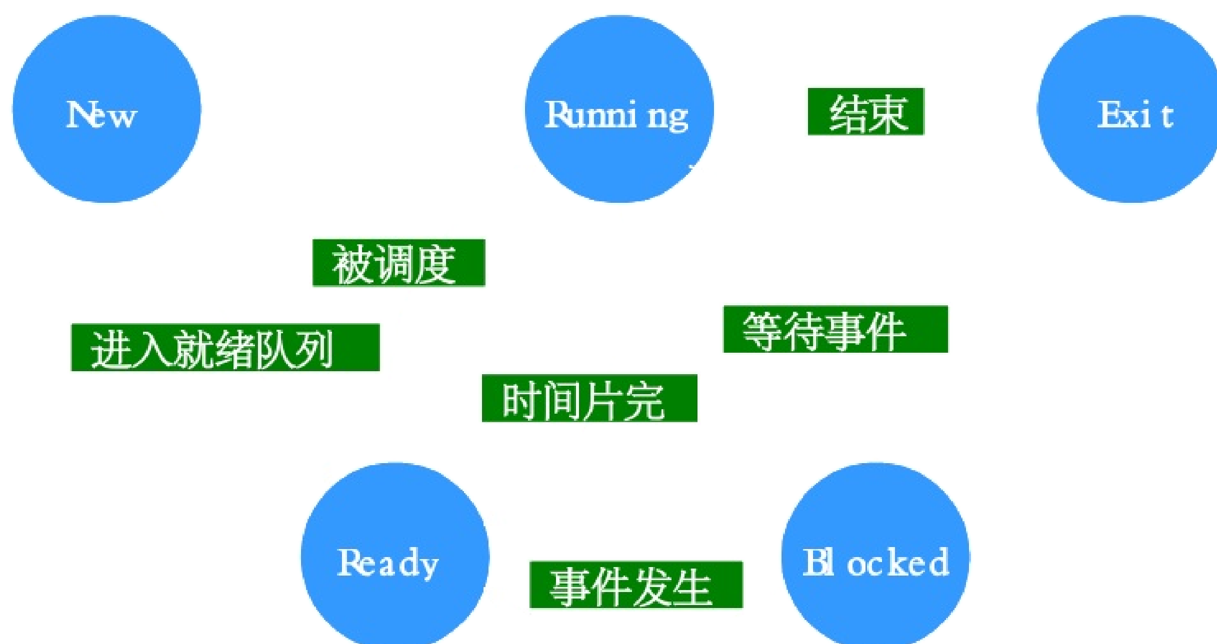
可以把执行应用程序的代码一直到执行操作系统中的进程切换处为止都认为是一个应用程序的执行过程（其中有操作系统的部分代码执行过过程）即进程。因为在这个过程中，没有更换到另外一个进程控制块的进程的页表和相关硬件寄存器。

从指令执行的角度看，如果再仔细分析一下操作系统这个软件的特点并细化一下进入内核原因，就可以看出进一步进行划分。操作系统的主要功能是给上层应用提供服务，管理整个计算机系统中的资源。所以操作系统虽然是一个软件，但其实是一个基于事件的软件，这里操作系统需要响应的事件包括三类：外设中断、CPU执行异常（比如访存错误）、陷入（系统调用）。如果用户进程通过系统调用要求操作系统提供服务，那么用户进程的角度看，操作系统就是一个特殊的软件库（比如相对于用户态的libc库，操作系统可看作是内核态的libc库），完成用户进程的需求，从执行逻辑上看，是用户进程“主观”执行的一部分，即用户进程“知道”操作系统要做的事情。那么在这种情况下，进程的代码空间包括用户态的执行程序和内核态响应用户进程通过系统调用而在核心特权态执行服务请求的操作系统代码，为此这种情况下的进程的内存虚拟空间也包括两部分：用户态的虚地址空间和核心态的虚地址空间。但如果此时发生的事件是外设中断和CPU执行异常，虽然CPU控制权也转入到操作系统中的中断服务例程，但这些内核执行代码执行过程是用户进程“不知道”的，是另外一段执行逻辑。那么在这种情况下，实际上是执行了两段目标不同的执行程序，一个是代表应用程序的用户进程，一个是代表中断服务例程处理外设中断和CPU执行异常的内核线程。这个用户进程和内核线程在产生中断或异常的时候，CPU硬件就完成了它们之间的指令流切换。

用户进程的运行状态分析

用户进程在其执行过程中会存在很多种不同的执行状态，根据操作系统原理，一个用户进程一般的运行状态有五种：创建（new）态、就绪（ready）态、运行（running）态、等待（blocked）态、退出（exit）态。各个状态之间会由于发生了某事件而进行状态转换。进程的状态转换图如下所示：

状态变化图



但在用户进程的执行过程中，具体在哪个时间段是出于上述状态的呢？上述状态是如何转变的呢？首先，我们看创建（new）态，操作系统完成进程的创建工作，而体现进程存在的就是进程控制块，所以一旦操作系统创建了进程控制块，则可以认为此时进程就已经存在了，但由于进程能够运行的各种资源还没准备好，所以此时的进程处于创建（new）态。创建了进程控制块后，进程并不能就执行了，还需准备好各种资源，如果把进程执行所需要的虚拟内存空间，执行代码，要处理的数据等都准备好了，则此时进程已经可以执行了，但还没有被操作系统调度，需要等待操作系统选择这个进程执行，于是把这个做好“执行准备”的进程放入到一个队列中，并可以认为此时进程处于就绪（ready）态。当操作系统的调度器从就绪进程队列中选择了一个就绪进程后，通过执行进程切换，就让这个被选上的就绪进程执行了，此时进程就处于运行（running）态了。到了运行态后，会出现三种事件。如果进程需要等待某个事件（比如主动睡眠10秒钟，或进程访问某个内存空间，但此内存空间被换出到硬盘swap分区中了，进程不得不等待操作系统把缓慢的硬盘上的数据重新读回到内存中），那么操作系统会把CPU给其他进程执行，并把进程状态从运行（running）态转换为等待（blocked）态。如果用户进程的应用程序逻辑流程执行结束了，那么操作系统会把CPU给其他进程执行，并把进程状态从运行（running）态转换为退出（exit）态，并准备回收用户进程占用的各种资源，当把表示整个进程存在的进程控制块也回收了，这进程就不存在了。在这整个回收过程中，进程都处于退出（exit）态。2考虑到在内存中存在多个处于就绪态的用户进程，但只有一个CPU，所以为了公平起见，每个就绪态进程都只有有限的时间片段，当一个运行态的进程用完了它的时间片段后，操作系统会剥夺此进程的CPU使用权，并把此进程状态从运行（running）态转换为就绪（ready）态，最后把CPU给其他进程执行。如果某个处于等待（blocked）态的进程所等待的事件产生了（比如睡眠时间到，或需要访问的数据已经从硬盘换入到内存中），则操作系统会通过把等待此事件的进程状态从等待（blocked）态转到就绪（ready）态。这样进程的整个状态转换形成了一个有限状态自动机。

有了上述对用户进程的特征分析后，接下来我们就通过跟踪用户进程的整个生命周期来阐述用户进程管理的设计与实现。

创建用户进程

在proj10中，我们已经完成了对内核线程的创建，但与用户进程的创建过程相比，创建内核线程的过程还远远不够。而这两个创建过程的差异本质上就是用户进程和内核线程的差异决定的。

应用程序的组成和编译

首先，我们要有一个应用程序，这里我们假定是hello应用程序，在user/hello.c中实现，代码如下：

```
#include <stdio.h>
#include <ulib.h>

int
main(void) {
    cprintf("Hello world!!.\n");
    cprintf("I am process %d.\n", getpid());
    cprintf("hello pass.\n");
    return 0;
}
```

hello应用程序只是输出一些字符串，并通过系统调用sys_getpid（在getpid函数中调用）输出代表hello应用程序执行的用户进程的进程标识--pid。

首先，我们需要了解ucore操作系统如何能够找到hello应用程序。这需要分析ucore和hello是如何编译的。修改Makefile，把第六行注释掉。然后在proj10.1目录下执行make，可得到如下输出：

```
……
+ cc user/hello.c

gcc -Iuser/ -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc  -fno-stack-protector -Ili
bs/ -Iuser/include/ -Iuser/libs/ -c user/hello.c -o obj/user/hello.o

ld -m    elf_i386 -nostdlib -T tools/user.ld -o obj/__user_hello.out  obj/user/libs/in
itcode.o obj/user/libs/panic.o obj/user/libs/stdio.o obj/user/libs/syscall.o obj/user/
libs/ulib.o obj/user/libs/umain.o  obj/libs/hash.o obj/libs/printfmt.o obj/libs/rand.o
 obj/libs/string.o obj/user/hello.o
……
ld -m    elf_i386 -nostdlib -T tools/kernel.ld -o bin/kernel  obj/kern/init/entry.o ob
j/kern/init/init.o …… -b binary …… obj/__user_hello.out
……
```

从中可以看出，hello应用程序不仅仅是hello.c，还包含了支持hello应用程序的用户态库：

- user/libs/initcode.S：所有应用程序的起始用户态执行地址“_start”，调整了EBP和ESP后，调用umain函数。
- user/libs/umain.c：实现了umain函数，这是所有应用程序执行的第一个C函数，它将调用应用程序的main函数，并在main函数结束后调用exit函数，而exit函数最终将调用sys_exit系统调用，让操作系统回收进程资源。
- user/libs/ulib.[ch]：实现了最小的C函数库，除了一些与系统调用无关的函数，其他函数是对访问系统调用的包装。
- user/libs/syscall.[ch]：用户层发出系统调用的具体实现。
- user/libs/stdio.c：实现cprintf函数，通过系统调用sys_putc来完成字符输出。
- user/libs/panic.c：实现**panic/**warn函数，通过系统调用sys_exit完成用户进程退出。

除了这些用户态库函数实现外，还有一些libs/*.[ch]是操作系统内核和应用程序共用的函数实现。这些用户库函数其实在本质上与UNIX系统中的标准libc没有区别，只是实现得很简单，但hello应用程序的正确执行离不开这些库函数。

【注意】**libs/*.[ch]*、*user/libs/*.[ch]、user/*.[ch]**的源码中没有任何特权指令。

在make的最后一步执行了一个ld命令，把hello应用程序的执行码obj/**user_hello.out**连接在了**ucore kernel**的末尾。且**ld**命令会在**kernel**中会把user_hello.out的位置和大小记录在全局变量_binary_obj***user_hello_out_start***和***_binary_obj***user_hello_out_size中，这样这个hello用户程序就能够和ucore内核一起被 bootloader 加载到内存里中，并且通过这两个全局变量定位hello用户程序执行码的起始位置和大小。而到了lab6的实验后，ucore会提供一个简单的文件系统，那时所有的用户程序就都不再用这种方法进行加载了。

用户进程的虚拟地址空间

在tools/user.ld描述了用户程序的用户虚拟空间的执行入口虚拟地址：

```
SECTIONS {
    /* Load programs at this address: "." means the current address */
    . = 0x800020;
```

在tools/kernel.ld描述了操作系统的内核虚拟空间的起始入口虚拟地址：

```
SECTIONS {
    /* Load the kernel at this address: "." means the current address */
    . = 0xC0100000;
```

这样ucore把用户进程的虚拟地址空间分了两块，一块与内核线程一样，是所有用户进程都共享的内核虚拟地址空间，映射到同样的物理内存空间中，这样在物理内存中只需放置一份内核代码，使得用户进程从用户态进入核心态时，内核代码可以统一应对不同的内核程序；另外一块是用户虚拟地址空间，虽然虚拟地址范围一样，但映射到不同且没有交集的物理内存空间中。这样当ucore把用户进程的执行代码（即应用程序的执行代码）和数据（即应用程序的全局变量等）放到用户虚拟地址空间中时，确保了各个进程不会“非法”访问到其他进程的物理内存空间。 这样ucore给一个用户进程具体设定的虚拟内存空间（kern/mm/memlayout.h）如下所示：

```
Virtual memory map:                                          Permissions
                                                             kernel/user

    4G ------------------> +---------------------------------+
                           |                                 |
                           |         Empty Memory (*)        |
                           |                                 |
                           +---------------------------------+ 0xFB000000
                           |   Cur. Page Table (Kern, RW)    | RW/-- PTSIZE
    VPT -----------------> +---------------------------------+ 0xFAC00000
                           |        Invalid Memory (*)       | --/--
    KERNTOP -------------> +---------------------------------+ 0xF8000000
                           |                                 |
                           |    Remapped Physical Memory     | RW/-- KMEMSIZE
                           |                                 |
    KERNBASE ------------> +---------------------------------+ 0xC0000000
                           |        Invalid Memory (*)       | --/--
    USERTOP -------------> +---------------------------------+ 0xB0000000
                           |           User stack            |
                           +---------------------------------+
                           |                                 |
                           :                                 :
                           |         ~~~~~~~~~~~~~~~~        |
                           :                                 :
                           |                                 |
                           ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                           |       User Program & Heap       |
    UTEXT ---------------> +---------------------------------+ 0x00800000
                           |        Invalid Memory (*)       | --/--
                           |  - - - - - - - - - - - - - - -  |
                           |    User STAB Data (optional)    |
    USERBASE, USTAB------> +---------------------------------+ 0x00200000
                           |        Invalid Memory (*)       | --/--
    0 -------------------> +---------------------------------+ 0x00000000
```

画一幅图，表示用户进程的虚拟地址空间和物理地址空间的映射关系和空间布局

创建并执行用户进程

在确定了用户进程的执行代码和数据，以及用户进程的虚拟空间布局后，我们可以来创建用户进程了。在proj10.1中第一个用户进程是由第二个内核线程initproc通过把hello应用程序执行码覆盖到initproc的用户虚拟内存空间来创建的，相关代码如下所示：

```
// kernel_execve - do SYS_exec syscall to exec a user program called by user_main kern
el_thread
static int
kernel_execve(const char *name, unsigned char *binary, size_t size) {
    int ret, len = strlen(name);
    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL), "0" (SYS_exec), "d" (name), "c" (len), "b" (binary), "D" (s
ize)
        : "memory");
    return ret;
}

#define __KERNEL_EXECVE(name, binary, size) ({                          \
            cprintf("kernel_execve: pid = %d, name = \"%s\".\n",        \
                    current->pid, name);                                \
            kernel_execve(name, binary, (size_t)(size));                \
        })

#define KERNEL_EXECVE(x) ({                                             \
            extern unsigned char _binary_obj___user_##x##_out_start[],  \
                _binary_obj___user_##x##_out_size[];                    \
            __KERNEL_EXECVE(#x, _binary_obj___user_##x##_out_start,     \
                            _binary_obj___user_##x##_out_size);         \
        })
……
// init_main - the second kernel thread used to create kswapd_main & user_main kernel
threads
static int
init_main(void *arg) {
#ifdef TEST
    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);
#else
    KERNEL_EXECVE(hello);
#endif
    panic("kernel_execve failed.\n");
    return 0;
}
```

对于上述代码，我们需要从后向前按照函数/宏的实现一个一个来分析。Initproc的执行主体是init_main函数，这个函数在缺省情况下是执行宏KERNEL_EXECVE(hello)，而这个宏最终是调用kernel_execve函数来调用SYS_exec系统调用，由于ld在链接hello应用程序执行码时定义了两全局变量：

- _binary_obj___user_hello_out_start：hello执行码的起始位置
- _binary_obj___user_hello_out_size中：hello执行码的大小

kernel_execve把这两个变量作为SYS_exec系统调用的参数，让ucore来创建此用户进程。当ucore收到此系统调用后，将依次调用如下函数

```
vector128(vectors.S)--> __alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)
--
-->syscall(syscall.c)-->sys_exec（syscall.c）-->do_execve(proc.c)
```

最终通过do_execve函数来完成用户进程的创建工作。此函数的主要工作流程如下：

- 首先为加载新的执行码做好用户态内存空间清空准备。如果mm不为NULL，则设置页表为内核空间页表，且进一步判断mm的引用计数减1后是否为0，如果为0，则表明没有进程再需要此进程所占用的内存空间，为此将根据mm中的记录，释放进程所占用户空间内存和进程页表本身所占空间。最后把当前进程的mm内存管理指针为空。由于此处的initproc是内核线程，所以mm为NULL，整个处理都不会做。
- 接下来的一步是加载应用程序执行码到当前进程的新创建的用户态虚拟空间中。这里涉及到读ELF格式的文件，申请内存空间，建立用户态虚存空间，加载应用程序执行码等。load_icode函数完成了整个复杂的工作。

load_icode函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。此函数有一百多行，完成了如下重要工作：

1. 调用mm_create函数来申请进程的内存管理数据结构mm所需内存空间，并对mm进行初始化；（mm_struct的介绍载第三章3.3.5小节）
2. 调用setup_pgdir来申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表（boot_pgdir所指）的内容拷贝到此新目录表中，最后让mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；
3. 根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并调用mm_map函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间；（vma_struct的介绍载第三章3.3.5小节）
4. 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；
5. 需要给用户进程设置用户栈，为此调用mm_mmap函数建立用户栈的vma结构，明确用户栈的位置在用户虚空间的顶端，大小为256个页，即1MB，但要注意，并没有给用户栈分配实际的物理内存；
6. 至此,进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；
7. 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断；

```
tf->tf_cs = USER_CS;
tf->tf_ds = USER_DS;
tf->tf_es = USER_DS;
tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;
```

至此，用户进程的用户环境已经搭建完毕。此时initproc将按产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”（位于trapentry.S的最后一句）后，将切换到用户进程hello的第一条语句位置_start处（位于user/libs/initcode.S的第三句）开始执行。

基于时间事件的等待与唤醒

Clock（时钟）中断(irq0，可回顾第二章2.4节) 可给操作系统提供有一定间隔的时间事件， 操作系统将其作为基本的计时单位，这里把两次时钟中断之间的时间间隔为一个时间片（timer splice）。基于此时间片，操作系统得以向上提供基于时间点的事件，并实现基于固定时间长度的等待和唤醒机制。在每个时钟中断发生时，操作系统可产生对应时间长度的时间事件，这样操作系统和应用程序可基于这些时间事件来构建基于时间的软件设计。在proj10.4中，实现了定时器timer的支持。

timer数据结构

sched.h定义了有关timer数据结构，

```
typedef struct {
    unsigned int expires;          //到期时间
    struct proc_struct *proc;     //等待时间到期的进程
    list_entry_t timer_link;      //链接到timer_list的链表项指针
} timer_t;
```

这几个成员变量描述了一个timer（定时器）涉及的相关因素，首先一个expires表明了这个定时器何时到期，而第二个成员变量描述了定时器到期后需要唤醒的进程，最后一个参数是一个链表项，用于把自身挂到系统的timer链表上，以便于扫描查找特定的timer。

timer相关操作

一个 timer在ucore中的生存周期可以被描述如下：

1. 某进程创建和初始化timer_t结构的一个timer，并把timer被加入系统timer管理列表timer_list中，进程设置为基于timer事件的阻塞态（即睡眠了），这样这个timer就诞生了，；
2. 系统时间通过时钟中断被不断累加，且ucore定期检查是否有某个timer的到期时间已经到了，如果没有到期，则ucore等待下一次检查，此timer会挂在timer_list上继续存在；
3. 如果到期了，则对应的进程又处于就绪态了，并从系统timer管理列表timer_list中移除该timer，自此timer就死亡退出了。

基于上述timer生存周期的流程，与timer相关的函数如下：

- timer init：对timer的成员变量进行初始化，设定了在expires时间之后唤醒proc进程
- add_timer：向系统timer链表timer_list添加某个初始化过的timer，这样该timer计时器将

在指定时间expires后被扫描到，如果等待则这个定时器timer的进程处在等待状态，并将将进程唤醒，进程将处于就绪态。
- del_timer：向系统timer链表timer_list删除（或者说取消）某一个计时器。该计时 器在取消后，对应的进程不会被系统在指定时刻expires唤醒。
- run_timer_list：被trap函数调用，遍历系统timer链表timer_list中的timer计时器，找出所有应该到时的timer计时器，并唤醒与此计时器相关的等待进程，再删除此timer计时器。在lab4/proj13以后，还增加了对进程调度器在时间事件产生后的处理函数的调用（在后续有进一步分析）。

有了这些函数的支持，我们就可以实现进程睡觉并被定时唤醒的功能了。比如ucore在用户函数库中提供了sleep函数，当用户进程调用sleep函数后，会进一步调用sys_sleep系统调用，在内核中完成sys_sleep系统调用服务的是do_sleep内核函数，其实现如下：

```
int
do_sleep(unsigned int time) {
……
    timer_t __timer, *timer = timer_init(&__timer, current, time);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_TIMER;
    add_timer(timer);
……
    schedule();
    del_timer(timer);
    return 0;
}
```

可以看出，do_sleep首先初始化了一个定时器timer，设置了timer的proc是当前进程，到期时间expires是参数time；然后把当前进程的状态设置为等待状态，且等待原因是等某个定时器到期；再调用schedule完成进程调度与切换，这时当前进程已经不占用CPU执行了。当定时器到期后，run_timer_list会删除timer且唤醒timer对应的当前进程，从而使得当前进程可以继续执行。

进程退出和等待进程

当进程执行完它的工作后，就需要执行退出操作，释放进程占用的资源。Ucore分了两步来完成这个工作，首先由进程本身完成大部分资源的占用内存回收工作，然后由此进程的父进程完成剩余资源占用内存的回收工作。为何不让进程本身完成所有的资源回收工作呢？这是因为进程要执行回收操作，就表明此进程还存在，还在执行指令，这就需要内核栈的空间不能释放，且表示进程存在的进程控制块不能释放。所以需要父进程来帮忙释放子进程无法完成的这两个资源回收工作。

为此在用户态的函数库中提供了exit函数，此函数最终访问sys_exit系统调用接口让操作系统来帮助当前进程执行退出过程中的部分资源回收。我们来看看ucore是如何做进程退出工作的。需要注意，这部分实现在proj10.2中才完成。所以我们这里是基于proj10.2的代码来进行分析。

首先，exit函数会把一个退出码error_code传递给ucore，ucore通过执行内核函数do_exit来完成对当前进程的退出处理，主要工作简单地说就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作，具体流程如下：

1. 如果current->mm != NULL，表示是用户进程，则开始回收此用户进程所占用的用户态虚拟内存空间；

   a. 首先执行“lcr3(boot_cr3)”，切换到内核态的页表上，这样当前用户进程目前只能在内核虚拟地址空间执行了，这是为了确保后续释放用户态内存和进程页表的工作能够正常执行；

   b. 如果当前进程控制块的成员变量mm的成员变量mm_count减1后为0（表明这个mm没有再被其他进程共享，可以彻底释放进程所占的用户虚拟空间了。），则开始回收用户进程所占的内存资源：

   - 调用exit_mmap函数释放current->mm->vma链表中每个vma描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后还把页表所占用的空间释放并把对应的页目录表项清空；
   - 调用put_pgdir函数释放当前进程的页目录所占的内存；
   - 调用mm_destroy函数释放mm中的vma所占内存，最后释放mm所占内存；

   c. 此时设置current->mm为NULL，表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕；

2. 这时，设置当前进程的执行状态current->state=PROC_ZOMBIE，当前进程的退出码current->exit_code=error_code。此时当前进程已经不能被调度了，需要此进程的父进程来做最后的回收工作（即回收描述此进程的内核栈和进程控制块）；

3. 如果当前进程的父进程current->parent处于等待子进程状态（即current->parent->wait_state==WT_CHILD），则唤醒父进程（即执行“wakup_proc(current->parent)”），让父进程帮助自己完成最后的资源回收工作；

4. 如果当前进程还有子进程，则需要把这些子进程的父进程指针设置为内核线程initproc，且各个子进程指针需要插入到initproc的子进程链表中。如果某个子进程的执行状态是PROC_ZOMBIE，则需要唤醒initproc来完成对此子进程的最后回收工作。

5. 执行schedule()函数，选择新的进程执行。

那么父进程如何完成对子进程的最后回收工作呢？这要求父进程要执行wait用户函数或wait_pid用户函数，这两个函数的区别是，wait函数等待任意子进程的结束通知，而wait_pid函数等待进程id号为pid的子进程结束通知。这两个函数最终访问sys_wait系统调用接口让ucore来完成对子进程的最后回收工作，即回收子进程的内核栈和进程控制块所占内存空间，具体流程如下：

1. 如果pid!=0，表示只找一个进程id号为pid的退出状态的子进程，否则找任意一个处于退出状态的子进程；
2. 如果此子进程的执行状态不为PROC_ZOMBIE，表明此子进程还没有退出，则当前进程只好设置自己的执行状态为PROC_SLEEPING，睡眠原因为WT_CHILD（即等待子进程退出），调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒，则重复跳回步骤1处执行；
3. 如果此子进程的执行状态为PROC_ZOMBIE，表明此子进程处于退出状态，需要当前进程（即子进程的父进程）完成对子进程的最终回收工作，即首先把子进程控制块从两个进程队列proc_list和hash_list中删除，并释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，消除了它所占用的所有资源。

【问题】哪些资源是子进程无法回收，需要父进程帮忙回收的？

【问题】当子进程执行了**sys_exit**系统调用，但父进程还没有执行**sys_wait**系统调用的情况下，子进程还能正常通知父进程让父进程回收子进程最后无法回收的子进程所占资源吗？

系统调用实现

系统调用的英文名字是System Call。操作系统为什么需要实现系统调用呢？其实这是实现了用户进程后，自然引申出来需要实现的操作系统功能。用户进程只能在操作系统给它圈定好的“用户环境”中执行，但“用户环境”限制了用户进程能够执行的指令，即用户进程只能执行一般的指令，无法执行特权指令。如果用户进程想执行一些需要特权指令的任务，比如通过网卡发网络包等，只能让操作系统来代劳了。于是就需要一种机制来确保用户进程不能执行特权指令，但能够请操作系统“帮忙”完成需要特权指令的任务，这种机制就是系统调用。

采用系统调用机制为用户进程提供一个获得操作系统服务的统一接口层，这样一来可简化用户进程的实现，把一些共性的、繁琐的、与硬件相关、与特权指令相关的任务放到操作系统层来实现，但提供一个简洁的接口给用户进程调用；二来这层接口事先可规定好，且严格检查用户进程传递进来的参数和操作系统要返回的数据，使得让操作系统给用户进程服务的同时，保护操作系统不会被用户进程破坏。

从硬件层面上看，需要硬件能够支持在用户态的用户进程通过某种机制切换到内核态。在第二章的2.4节和2.5节讲述中断硬件支持和软件处理过程其实就可以用来完成系统调用所需的软硬件支持。下面我们来看看如何在ucore中实现系统调用。

初始化系统调用对应的中断描述符

在ucore初始化函数kern_init中调用了idt_init函数来初始化中断描述符表。在proj10.1以前的一个proj4.4.1（其实proj4.4.1就是为proj10.1做准备的）中，为了实现了从用户态返回到内核态的功能，设置了一个特定中断号T_SWITCH_TOK的中断门，让用户态程序通过执行一个特殊的指令 “INT 中断号”来完成从用户态到内核态的切换，这个中断号就是T_SWITCH_TOK。proj10.1参考proj4.4.1的做法，首先要设置一个特定中断号的中断门，专门用于用户进程访问系统调用。此事由ide_init函数完成：

```
void
idt_init(void) {
    extern uintptr_t __vectors[];
    int i;
    for (i = 0; i < sizeof(idt) / sizeof(struct gatedesc); i ++) {
        SETGATE(idt[i], 1, GD_KTEXT, __vectors[i], DPL_KERNEL);
    }
    SETGATE(idt[T_SYSCALL], 1, GD_KTEXT, __vectors[T_SYSCALL], DPL_USER);
    lidt(&idt_pd);
}
```

在上述代码中，可以看到在执行加载中断描述符表lidt指令前，专门设置了一个特殊的中断描述符idt[T_SYSCALL]，它的特权级设置为DPL_USER，中断向量处理地址在__vectors[T_SYSCALL]处。这样建立好这个中断描述符后，一旦用户进程执行 “INT T_SYSCALL”后，由于此中断允许用户态进程产生（注意它的特权级设置为DPL_USER），所以CPU就会从用户态切换到内核态，保存相关寄存器，并跳转到__vectors[T_SYSCALL]处开始执行，形成如下执行路径：

```
vector128(vectors.S)--> __alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)
--
-->syscall(syscall.c)-
```

在syscall中，根据系统调用号来完成不同的系统调用服务。

建立系统调用的用户库准备

在操作系统中初始化好系统调用相关的中断描述符、中断处理起始地址等后，还需在用户态的应用程序中初始化好相关工作，简化应用程序访问系统调用的复杂性。为此在用户态建立了一个中间层，即简化的libc实现，在user/libs/ulib.[ch]和user/libs/syscall.[ch]中完成了对访问系统调用的封装。用户态最终的访问系统调用函数是syscall，实现如下：

```
static inline int
syscall(int num, ...) {
    va_list ap;
    va_start(ap, num);
    uint32_t a[MAX_ARGS];
    int i, ret;
    for (i = 0; i < MAX_ARGS; i ++) {
        a[i] = va_arg(ap, uint32_t);
    }
    va_end(ap);

    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a[0]),
          "c" (a[1]),
          "b" (a[2]),
          "D" (a[3]),
          "S" (a[4])
        : "cc", "memory");
    return ret;
}
```

从中可以看出，应用程序调用的exit/fork/wait/getpid等库函数最终都会调用syscall函数，只是调用的参数不同而已，如果看最终的汇编代码会更清楚：

```
……
  34:    8b 55 d4                 mov    -0x2c(%ebp),%edx
  37:    8b 4d d8                 mov    -0x28(%ebp),%ecx
  3a:    8b 5d dc                 mov    -0x24(%ebp),%ebx
  3d:    8b 7d e0                 mov    -0x20(%ebp),%edi
  40:    8b 75 e4                 mov    -0x1c(%ebp),%esi
  43:    8b 45 08                 mov    0x8(%ebp),%eax
  46:    cd 80                    int    $0x80
      48:    89 45 f0                 mov    %eax,-0x10(%ebp)
……
```

可以看到其实是把系统调用号放到EAX，其他5个参数a[0]~a[4]分别保存到EDX/ECX/EBX/EDI/ESI五个寄存器中，及最多用6个寄存器来传递系统调用的参数，且系统调用的返回结果是EAX。比如对于getpid库函数而言，系统调用号（SYS_getpid=18）是保存在EAX中，返回值（调用此库函数的的当前进程号pid）也在EAX中。

与用户进程相关的系统调用

在proj10.1中，与进程相关的各个系统调用属性如下所示：

系统调用名	含义	具体完成服务的函数
SYS_exit	process exit	do_exit
SYS_fork	create child process, dup mm	do_fork-->wakeup_proc
SYS_wait	wait child process	do_wait
SYS_exec	after fork, process execute a new program	load a program and refresh the mm
SYS_clone	create child thread	do_fork-->wakeup_proc
SYS_yield	process flag itself need resecheduling	proc->need_sched=1, then scheduler will rescheule this process
SYS_sleep	process sleep	do_sleep
SYS_kill	kill process	do_kill-->proc->flags |= PF_EXITING, -->wakeup_proc-->do_wait-->do_exit
SYS_getpid	get the process's pid	

通过这些系统调用，可方便地完成从进程/线程创建到退出的整个运行过程。

系统调用的执行过程

与用户态的函数库调用执行过程相比，系统调用执行过程的有四点主要的不同：

- 不是通过“CALL”指令而是通过“INT”指令发起调用；
- 不是通过“RET”指令，而是通过“IRET”指令完成调用返回；
- 当到达内核态后，操作系统需要严格检查系统调用传递的参数，确保不破坏整个系统的安全性；
- 执行系统调用可导致进程等待某事件发生，从而可引起进程切换；

下面我们以getpid系统调用的执行过程大致看看操作系统是如何完成整个执行过程的。当用户进程调用getpid函数，最终执行到“INT T_SYSCALL”指令后，CPU根据操作系统建立的系统调用中断描述符，转入内核态，并跳转到vector128处（kern/trap/vectors.S），开始了操作系统的系统调用执行过程，函数调用和返回操作的关系如下所示：

```
vector128(vectors.S)--> __alltraps(trapentry.S)-->trap(trap.c)-->trap_dispatch(trap.c)
--
-->syscall(syscall.c)-->sys_getpid(syscall.c)-->……-->__trapret(trapentry.S)
```

在执行trap函数前，软件还需进一步保存执行系统调用前的执行现场，即把与用户进程继续执行所需的相关寄存器等当前内容保存到当前进程的中断帧trapframe中（注意，在创建进程是，把进程的trapframe放在给进程的内核栈分配的空间的顶部）。软件做的工作在vector128和__alltraps的起始部分：

```
vectors.S::vector128起始处:
  pushl $0
  pushl $128
......
trapentry.S::__alltraps起始处:
pushl %ds
  pushl %es
  pushal
……
```

自此，用于保存用户态的用户进程执行现场的trapframe的内容填写完毕，操作系统可开始完成具体的系统调用服务。在sys*getpid*函数中，简单地把当前进程的*pid*成员变量做为函数返回值就是一个具体的系统调用服务。完成服务后，操作系统按调用关系的路径原路返回到_alltraps中。然后操作系统开始根据当前进程的中断帧内容做恢复执行现场操作。其实就是把trapframe的一部分内容保存到寄存器内容。恢复寄存器内容结束后，调整内核堆栈指针到中断帧的tf_eip处，这是内核栈的结构如下：

```
/* below here defined by x86 hardware */
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
/* below here only when crossing rings */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
```

这时执行“IRET”指令后，CPU根据内核栈的情况回复到用户态，并把EIP指向tf_eip的值，即“INT T_SYSCALL”后的那条指令。这样整个系统调用就执行完毕了。

基于内核线程实现全局内存页替换机制

实验目标

到proj11为止，还没有能够在ucore中实现一个完整的内存页替换机制。但其实在lab2的proj8中，已经为ucore实现内存页替换机制提供了大量的支持，并在相关测试函数kern/mm/swap.c::check_swap中进行了检查。但这个检查只是说明了proj8提供了能够完成内存页替换机制的数据结构和函数支持，即已经一砖一瓦地完成了门窗、墙壁等建筑工作，还差把相关部件完整组织起来实现成一个完整的房子。proj11就是完成这最后一步，采用内核线程来实现内存页替换机制，使得用户进程在快用完内存后，可以通过内存页替换机制把不常用的页换出到硬盘swap分区中，常用的页保存在内存中，保持系统中有足够的内存给用户进程使用。

proj11概述

实现描述

proj11是lab3的第六个project。它在proj10.4的基础上实现了基于内核线程的内存页替换机制，主要扩展设计了专门用于执行内存页替换的内核线程kswapd，并增加了等待队列、扩展了进程控制块的成员变量mm的等，使得在用户进程申请存不足或系统空闲内存不足的情况下，通过执行kswapd内存线程，实现内存页替换，把不常用的页放到硬盘swap分区上，给系统提供足够的空闲空间。

项目组成

```
proj11
├── ……
|   ├── mm
|   |   ├── ……
|   |   ├── pmm.c
|   |   ├── swap.c
|   |   ├── swap.h
|   |   ├── vmm.c
|   |   └── vmm.h
|   ├── process
|   |   ├──……
|   |   ├── proc.c
|   |   └── proc.h
|   └── sync
|         ├── ……
|         ├── wait.c
|         └── wait.h
└── user
    ├── cowtest.c
    ├── swaptest.c
    └── ……

17 directories, 114 files
```

相对于proj10.4，proj11在内核方面主要增加了有关kswapd内核线程和相关函数以及等待队列实现，在用户程序方面，增加测试ucore的COW实现的用户程序cowtest.c和测试内存页置换实现的swaptest.c。主要修改和增加的文件如下：

- kern/mm/pmm.c：扩展了alloc_pages函数，使得它能够在没有获得所需空闲内存页后，进一步调用tre_free_pages来要求ucore释放足够的空闲页，从而再次要求所需空闲页，直到要求得到满足为止。
- kern/mm/swap.[ch]：更新try_free_pages的实现，完成让当前进程睡眠，并唤醒kswapd内核线程，让它完成对空闲页的回收。同时实现了内核线程kswapd的执行主体kswapd_main函数，此函数完成具体的内存页置换机制。
- kern/sync/wait.[ch]：实现等待队列机制，使得内存页等资源无法得到满足的进程能够处于等待状态，并在资源得到满足后让进程继续执行。
- kern/mm/vmm.[ch]：扩展了mm_struct结构，并修改相关函数，使得所有进程的成员变量mm能够链入到全局mm_struct结构的链表proc_mm_list中。

编译运行

编译并运行proj11的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
(THU.CST) os is loading ...

Special kernel symbols:
……
check_vmm() succeeded.
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
check_swap() succeeded.
……
++ setup timer interrupts
kernel_execve: pid = 3, name = "swaptest".
buffer size = 00500000
parent init ok.
child 9 fork ok, pid = 13.
child 8 fork ok, pid = 12.
child 7 fork ok, pid = 11.
child 6 fork ok, pid = 10.
child 5 fork ok, pid = 9.
child 4 fork ok, pid = 8.
child 3 fork ok, pid = 7.
child 2 fork ok, pid = 6.
child 1 fork ok, pid = 5.
child 0 fork ok, pid = 4.
check cow ok.
round 0
round 1
round 2
round 3
round 4
child check ok.
wait ok.
check buffer ok.
swaptest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:430:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

表面上看不出上述输出对内存页置换算法实现的具体体现。不过通过Makefile和对swaptest.c程序的分析，还是能够看出proj11的执行与其他进程的执行不同：

```
Makefile:
……
QEMUOPTS = -m 48m -hda $(UCOREIMG) -drive file=$(SWAPIMG),media=disk,cache=writeback
swaptest.c
……
const int size = 5 * 1024 * 1024;
char *buffer;
……
main(void){
……
(buffer = malloc(size))
……
    for (i = 0; i < pids; i ++) {
        if ((pid[i] = fork()) == 0) {
……
}
```

通过Makefile，可以看到qemu只模拟出了48MB的物理内存空间，但swaptest.c创建了10个子进程，且每个子进程都会复制全局变量buffer，且会对buffer中的所有元素进行写操作。由于每个buffer的空间大小为5MB，所以10个子进程和1个父进程的buffer所占虚拟空间总和为55MB，大于实际的48MB物理内存空间。而在操作系统设计上，用户进程的用户空间是没必要都保存在内存中，这使得必须把某些页换出到硬盘swap分区才能确保所有子进程都能正常执行完毕。为此，我们还需进一步分析proj11中ucore具体的内存页置换机制的实现和执行过程。下面将从实现方面对此进行进一步阐述。

等待队列设计与实现

为了支持用户进程完成特定事件的等待和唤醒操作，ucore设计了等待队列，从而使得用户进程可以方便地实现由于某事件没有完成而睡眠，并且在事件完成后被唤醒的整个操作过程。

其基本设计思想是：当一个进程由于某个事件没有产生而需要在某个睡眠等待时，设置自身运行状态为PROC_SLEEPING，等待原因为某事件，然后将自己的进程控制块指针和等待标记组装到一个数据结构为wait_t的等待项数据中，并把这个等待项的挂载到等待队列wait_queue的链表中，再执行schedule函数完成调度切换；当某些事件发生后，另一个任务（进程）会唤醒等待队列wait_queue上的某个或者所有进程，唤醒操作就是将等待队列wait_queue中的等待项中的进程运行状态设置为可调度的状态，并且把等待项从等待队列中删除。下面是等待队列的设计与实现分析。

数据结构描述

等待项的定义：

```
typedef struct {
    struct proc_struct *proc;
    uint32_t wakeup_flags;
    wait_queue_t *wait_queue;
    list_entry_t wait_link;
} wait_t;
```

这里等待项的成员变量proc表明了等待某事件的进程控制块指针，wakeup_flags是唤醒进程的事件标志（多个标志可以有逻辑或的关系，形成复合事件标志），wait_queue是此等待项所属的等待队列，wait_link用于链接到等待队列wait_queue中。

等待队列的定义：

```
typedef struct {
    list_entry_t wait_head;
} wait_queue_t;
```

等待队列就是一个双向链表的头指针。

等待队列相关操作函数

初始化

如果要使用等待队列，首先需要声明并初始化等待队列。以proj11为例，在kern/mm/swap.c中有一个等待队列的变量声明和在swap_init函数中执行的对应初始化：

```
static wait_queue_t kswapd_done;
…
wait_queue_init(&kswapd_done);
```

执行等待

如果某进程需要等待某事件，则需要设置自己的运行状态为PROC_SLEEPING，构建并初始化一个等待项，再挂入到某个等待队列中。以proj11为例，某进程申请内存资源无法满足，需要等待kswapd内核线程给系统更多的内核资源，于是在try_free_pages函数中执行了如下操作：

```
wait_t __wait, *wait = &__wait;
wait_init(wait, current);
        current->state = PROC_SLEEPING;
        current->wait_state = WT_KSWAPD;
        wait_queue_add(&kswapd_done, wait);
```

这里可以看到，首先声明了一个等待项__wait，然后调用wait_init函数对此等待项进行了初始化；并进一步把当前进程的运行状态设置为PROC_SLEEPING，睡眠原因设置为WT_KSWAPD，即等待kswapd释放出更多的空闲内存；最后把此等待项加入到等待队列kwapd_done中。

执行唤醒

当某个事件产生后，需要唤醒等待在等待队列中的睡眠进程。以proj11为例，当kswapd内核线程释放出更多的空闲内存后，就需要唤醒等待更多内存的进程，在kswapd内核线程的主体执行函数kswapd_main中调用了

```
kswapd_wakup_all函数：
wakeup_queue(&kswapd_done, WT_KSWAPD, 1);
```

这个函数就完成了唤醒功能，它会遍历kswapd_done等待队列上的所有等待项，找到一个就执行wakeup_wait函数，来进一步调用wakup_proc函数来唤醒挂在等待项上的睡眠进程。

上面是使用等待队列的基本流程。为了能够更好地完善整个基于等待队列的等待唤醒机制，在wait.[ch]中提供了一系列函数：

- void wait_init：初始化等待项
- void wait_queue_init：初始化等待队列

- void wait_queue_add：把一个等待项加入到一个等待队列中
- void wait_queue_del：从一个等待队列中删除一个等待项
- wait_t *wait_queue_next：查找挂在某等待队列中的等待项指向的下一个等待项
- wait_t *wait_queue_prev：查找挂在某等待队列中的等待项指向的前一个等待项
- wait_t *wait_queue_first：查找挂在某等待队列中的第一个等待项
- wait_t *wait_queue_last：查找挂在某等待队列中的最后一个等待项
- bool wait_queue_empty：判断等待队列是否为空
- bool wait_in_queue：单品某等待项是否在等待队列中
- void wakeup_wait：唤醒等待项中的睡眠进程，删除等待队列中的等待项（参数del确定是否删除）
- void wakeup_first：唤醒等待队列中第一个等待项中的睡眠进程，删除等待队列中的这个等待项（参数del确定是否删除）
- void wakeup_queue：唤醒等待队列中所有等待项中的睡眠进程，删除等待队列中的对应等待项（参数del确定是否删除）

内存页置换机制的执行过程

在lab2/proj8中已经完成了大部分内存页置换所需的功能函数，但还没有有机地整和在ucore中，成为ucore内存管理子系统的组成部分。当有了内核线程机制后，我们就可以把内存页置换的功能整合到一个内核线程中，从而实现一个内存页置换线程kswapd，专门负责完成内存页置换的工作。

创建kswapd内核线程

在第1个内核线程initproc的主体执行函数init_main中，完成了对kswapd内核线程的创建工作：

```
static intinit_main(void *arg) {    int pid;    if ((pid = kernel_thread(kswapd_main,
NULL, 0)) <= 0) {        panic("kswapd init failed.\n");    }    kswapd = find_proc(pi
d);    set_proc_name(kswapd, "kswapd");
……
```

并且保存了创建用户进程前剩余页的情况和已经分配的slab的容量计数，这样在接下来创建完用户进程，并且所有用户进程执行完毕后，再判断所有进程结束后的剩余页的情况和已经分配的slab的容量计数，如果二者相等，这表明内存管理功能基本正确。

```
size_t nr_free_pages_store = nr_free_pages();  size_t slab_allocated_store = slab_allo
cated();
……
cprintf("all user-mode processes have quit.\n");
……    assert(nr_free_pages_store == nr_free_pages());    assert(slab_allocated_store =
= slab_allocated());    cprintf("init check memory pass.\n");    return 0;
```

触发kswapd内核线程

ucore目前大致有两种触发kswapd内核线程的策略，即积极策略和消极策略。积极策略是指ucore周期性地（或在系统不忙的时候）唤醒kswapd内核线程，让它主动把某些认为“不常用”的页换出到硬盘上，从而确保系统中总有一定数量的空闲页存在，这样当需要空闲页时，基本上能够及时满足需求；消极换出策略是指，ucore试图得到空闲页时，发现当前没有空闲的物理页可供分配，这时才唤醒kswapd内核线程，让kswapd开始查找“不常用”页面，并把一个或多个这样的页换出到硬盘上。

对于积极策略，即每隔1秒唤醒一次内核线程kswapd。在实现上是基于定时器的触发，即在kswapd的主体执行函数kswapd_main的末尾执行了“do_sleep(1000)；”，这表明了每隔1秒，kswapd就要执行一个回收空闲线程的迭代执行过程。对于消极策略，则是在ucore调用alloc_pages函数获取空闲页时，此函数如果发现无法从页分配器获得空闲页，就会进一步调用try_free_pages来唤醒线程kswapd，让kswapd 换出某些页。在执行try_free_pages函数时，由于当前用户进程要求的空闲内存空间无法得到满足，所以首先让当前用户进程睡眠并加入到等待队列中，睡眠原因设置为WT_KSWAPD，即等待kswapd释放出更多的空闲内存，然后唤醒kswapd：

```
……
        wait_init(wait, current);        current->state = PROC_SLEEPING;        curren
t->wait_state = WT_KSWAPD;        wait_queue_add(&kswapd_done, wait);        if (kswap
d->wait_state == WT_TIMER) {            wakeup_proc(kswapd);
……
```

全局页面置换算法的数据结构设计

根据lab2的设计，我们可以知道ucore中表示内存中物理页使用情况的变量是基于数据结构Page的全局变量pages数组，pages的每一项表示了计算机系统中一个物理页的使用情况。如果一个物理页在硬盘上有一个页备份，则ucore需要记录在硬盘中页备份的位置，同时还需要记录swap 分区上的页备份的使用计数。

为了表示物理页可被换出或已被换出的情况，ucore对部分内存相关数据和数据结构进行了扩展。如果一个页被换出了，则页的内容保存在硬盘swap分区上有一个连续扇区中（称为一个swap page），而对应此页的页表项PTE的Present位为0，表示已经没有对应的物理内存页映射关系了，但如果高24不为0，则表示这个页虽然在物理内存中不存在了，但保存在swap分区中以PTE高24位为偏移位置的swap page中。这里我们把Present位为0且高24位不为0的PTE称为一个 swap entry。一个有效的swap entry对应着一个swap page。而且不同页表中的swap entry可对已一个swap page。

为了表示存储 swap 分区上的页的使用计数，在swap.c 里面声明了全局的 mem_map 数据结构。如果一个ucore在swap分区上分配了一个

swap.c 里还声明了两个链表，分别是 active_list 和 inactive_list，分别表示已经有对应swap page的且处于“活跃”状态/“不活跃”状态的物理页所形成的链表。所有已经有对应swap page的物理页page 必须处于两个链表中的一个。

为了更好地对应swap page/swap entry，在描述物理页的Page数据结构专门设立了几个与页面置换相关的成员变量：

```
struct Page {
    uint32_t flags;   // array of flags that describe the status of the page frame
    swap_entry_t index;      // stores a swapped-out page identifier
list_entry_t swap_link; // swap hash link
……
};
```

首先flag的含义做了扩展：

```
// the page is in the active or inactive page list (and swap hash table)
#define PG_swap     4
// the page is in the active page list
#define PG_active   5
```

Page结构中的index值表示物理页在硬盘中页备份的位置，它保存了被换出的页的页表项PTE（即swap entry）高24位的内容，即硬盘中对应页备份的起始扇区位置值（以扇区为单位）。如果 page数据结构的 flags 设置了PG_swap 为1，则表示该 page 中的 index 是有效的 swap entry的索引值，从而该物理页上的数据可以被写出到 index 所表示的 swap page 上去。

Page结构中的swap_link保存了以entry为hash索引的链表项，这样根据entry，就可以快速的对 page 数据结构进行查找。但hash数组在哪里呢？对于在硬盘上有页备份的物理页（简称swap_page），需要统一管理起来，为此在proj8中就增加了全局变量：

```
static list_entry_t hash_list[HASH_LIST_SIZE];
```

hash*list*数组就是我们需要的*hash*数组，根据 *index(swap entry)* 索引全部的 *swap* page 的指针，这样通过hash函数

```
#define entry_hashfn(x)                 (hash32(x, HASH_SHIFT))
```

可以快速地根据entry找到对应的page 数据结构。

正如页替换算法描述的那样，把“常用”的可被换出页和“不常用”的可被换出页分别集中管理起来，形成 active_list 和 inactive_list 两个链表。 ucore的页面置换算法会根据相应的准则把它认为“常用”的物理页放到active_list链表中，而把它认为“不常用”的物理页放到inactive_list链表中。一个标记了 PG_swap 的页总是需要在这两个链表之间移动。

前面介绍过 mem_map 数组，他是用来记录 swap_page 的引用次数的。因为 swap 分区上的 swap page 实际上是某个物理页的数据备份。所以，一个物理页page的 page_ref 与 对应 swap page 的 mem_map[offset] (offset = swap_offset(entry)) 值的和是这个页的真实引用计数。page_ref 表示PTE对该物理页的映射的个数；mem_map 表示 PTE对该 swap 备份页的

映射个数。当 page_ref 为0 的时候，表示物理页可以被回收；当 mem_map[offset] 为0的时候，表示 swap page 可以被重新存储其他的物理页内容（前面介绍过，可以回收，但是在万不得已的情况下才眞的回收）。

【注意】

ucore 目前使用的 PIO 的方式读写 IDE 磁盘，这样的好处是，磁盘读入、写出操作可以认为是同步的，即当前CPU需要等待磁盘读写完毕后再进行进一步的工作。由于磁盘操作相对CPU的速度而言是很慢的，这使得会浪费大量的 CPU 时间在等IO操作上。于是我们总是希望能够在 IO 性能上有更大的提升，比如引入 DMA 这种异步的 IO 机制，为了避免后续开发上的各种不便和冲突，我们假设所有的磁盘操作都是异步的（也包括后面的实验），即使目前是通过 PIO 完成的。

假定某page的 flags中的PG_swap 标志位为 1，并且PG_active标志位也为 1，则表示该page 在 swap 的 active_list 中，否则在 inactive_list 中。 active_list 中的页表示活跃的物理页，即页表中可能存在多个 PTE 指向该物理页（这里可以是同一个页表中的多个 entry，在后面lab3的实验里面有了进程以后，也可以是多个进程的页表的多个 entry）；反过来，inactive_list 链表所链接的 page 通常是指没有 PTE 再指向的页。

【注意】

需要强调两点设计因素：

1. 一个 page 是在 active_list 还是在 inactive_list 的条件不是绝对的；
2. 只有 inactive_list 上的页才会被尝试换出。

这两个设计因素的设计起因如下：

1. 我们知道一个 page 换出的代价是很大的（磁盘操作），并且我们假设所有的磁盘操作都是异步的，那么换出一个 active 的页就变得非常不值得。因为在还有多个 PTE 指向他情况下进行换出操作（异步 IO 可能导致进程切换）的较长过程中，这个页可以随时被其它进程写脏。而硬件提供给内核的接口（即页表项PTE的dirty位）使得内核只能知道一个页是否是脏的（不能明确知道一个页的哪个部分是脏的），当这种情况发生时，就导致了一次无效的写出。
2. active_list 和 inactive_list 的维护只能由 与swap有关的集中操作来完成。特别是在lab3/proj11引入 kswapd内核线程之后，所有的内存页换出任务都交给kswapd，这样减少了复杂的同步互斥实现（在lab5中会重点涉及）。
3. 页面换入换出有关的操作需要做的就是尽可能的完成如下三件事情：
    - 将 PG_swap 为 0 的页转变成 PG_swap 为1 的页。即尽可能的给每个物理页分配一个 swap entry（当然前提是有足够大的 swap 分区）。
    - 将页从 active_list 上移动到 inactive_list 上。如果一个页还在 active_list 上，说明还有 PTE 指向此“活跃”的物理页。所以需要在完成内存页换出时断开对这些物理页的引用，把它变成不活跃的（inactive）。只有把所有的PTE对某page的引用都断开

（即 page的page_ref 为0）后，就可以将此page从 active_list 移动到 inactive_list 上。
- 将 inactive_list 上的页写出并释放掉。inactive_list 上的page表示已没有 PTE 指向此page了，那么该page可以被释放，如果该page被写过，那还需把此page换出到swap分区上。如果在整个换出过程（异步 IO）中没有其他进程再写这个物理页（即没有 PTE 在引用它或有PTE引用但页没有写脏），就认为这个物理页是可以安全释放的了。那么将它从 inactive_list 上取下，并调用 page_free函数 实现 page 的回收。

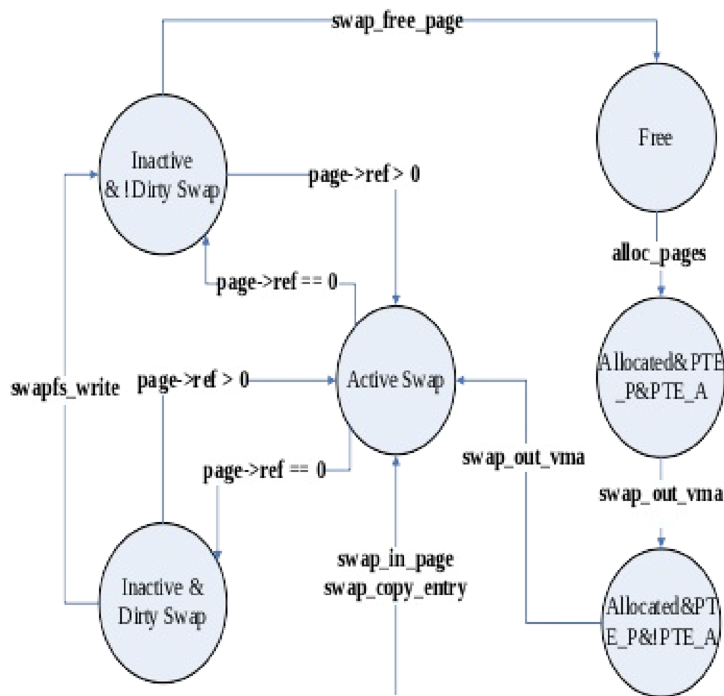
4. 值得注意的是，内存页换出操作只有特定时候才被调用，即通过 执行try_free_pages函数或者定时器机制（在lab3/proj10.4才引入）定期唤醒kswapd内核线程。这样会导致内存页换出操作对两个链表上的数据都不够敏感。比如处于 active_list 上的 page，可能在 kswapd 工作的时候，已经没有 PTE 再引用它了；再如相应的进程退出了，并且相应的地址空间已经被内核回收，从而变成了一个 inactive 的 page；还存在inactive_list 上的 page 也可能在换出的时候，其它进程通过 page fault，又将 PTE 指向他，进而变成一个实际上 active 的页。所以说，active 和 inactive 条件并不绝对。

全局页面置换算法的执行逻辑

其实在lab2/proj8中并没有完全实现页面置换算法，只是实现了其中的部分关键函数，并通过check_swap来验证了这些函数的正确性。直到lab3的proj11才形成了完整的页面置换逻辑，而这个页面置换逻辑基本上是改进的时钟算法的一个实际扩展版本。

页状态变化关系

ucore 采用的页面置换算法是一个全局的页面置换算法，因为它收集了ucore中所有用户态进程（这里可理解为ucore中运行的每个用户态程序）的可换出页，并把这些可换出页中的一部分转换为空闲页。其次它考虑了页的访问情况（根据PTE中PTE_A位的值）和读写情况（根据PTE中PTE_D位的值）。如果页被访问过，则把PTE_A位清零继续找下一页；如果页没有被访问过，这此页就成为了active状态的可换出页，并放入active_list链表中，这时需要把对应的PTE转换成为一个swap entry(高24位保存为硬盘缓存页的起始扇区号，PTE_P位清零)；接着refill_inactive_scan函数会把处于active状态的部分可换出页转换成inactive状态，并放入inactive_list链表中；然后page_launder函数扫描inactive_list中的处于inactive状态的可换出页，如果此页不是dirty的，则把它直接转换成空闲页，如果此页是dirty的，则执行换出操作，把该页换出到硬盘上保存。这个页的状态变化图如下图所示。



ucore的物理页状态变化图

确定腾出的空闲页数和断开的PTE映射数

在proj11中同时实现了积极换出策略和消极换出策略，这都是通过在不同的时机执行kwapd_main函数来完成的。当ucore调用alloc_pages函数以获取空闲页，但物理页内存分配器无法满足请求时，alloc_pages函数将调用try_free_pages来通过直接唤醒内核线程kswapd的方式执行kwapd_main函数，完成对页的换出操作和生成空闲页的操作。这是一种消极换出的策略。另外，ucore也设立了通过设置timer来唤醒线程的方式每秒执行一次kwapd_main函数，完成对页的换出操作和生成空闲页的操作。这是一种积极换出的策略。

如果alloc_pages 执行分配 n 连续物理页失败，则会通过调用tree_free_page来唤醒kswapd线程。kswapd需要尽可能的满足分配 n 个连续物理页的需求。既然需求是 n 个连续物理页，那么 kswapd 所需要释放的物理页就应该大于 n 个；每个页可能在某个或者许多个页表的不同的地方有 PTE 映射（特别是 copy on write 之后，这种情况更为普遍），那么 kswapd 所需要断开的 PTE 映射就远远不止 n 个。

Linux 实现了能够根据 physical address 在页表中快速定位的数据结构，但是实现起来过于复杂，这里 ucore 采用了一个比较笨的方法，即遍历所有存在的页表结构，断开足够多的 PTE 映射。这里足够多是个经验公式，采用 n\<\<5 。当然这也可能失败，那么 kswapd 就会尝试一定次数。当他实在无能为力的时候，也就放弃了。而 alloc_pages 也会不停的调用try_free_pages 进行尝试，当尝试不停的遭遇失败的时候，程序中会有许多句 warn 来输出这些调试信息。而 Linux 的方案是选择一个占用内存最多的进程杀掉并释放出资源，来尽可能的满足当前程序的需求（注意，这里当前程序是指内核服务或者调用），直到程序从内核态正常退出；ucore 的这种设计显然是相对简单了

页面置换大致流程

kswapd_main是ucore整个页面置换算法的总控部分，其大致思路是根据当前的空闲页情况查找出足够多的可换出页（swap page，即page的PG_swap标志位为1），然后根据这些可换出页的访问情况确定哪些是“常用”页（即page的PG_swap标志位和PG_active标志位为1），哪些是“不常用”页，最后把“不常用”的页转换成空闲页。思路简单，但具体实现相对复杂。kswapd腾出更多空闲页的核心是尽可能的断开页表中的 PTE 映射，其整个处理流程分为三步：

- 就是把尽可能多的 page（有对应合法PTE项，即对应PTE项的PTE_P标志位为1） 转变成 具有PG_active的page（此时的page PG_swap标志位和PG_active标志位都为1，但对应的PTE项的PTE_P标志位为0），并移动到 active list中；
- 接着把 active list 中的页（这些page的PG_swap标志位和PG_active标志位为1）尽可能多的变成 inactive list 中页（这些page的PG_swap标志位为1且PG_active标志位为0）；
- 最后把 inactive list 的页（这些page的PG_swap标志位为1且PG_active标志位为0）转换成空闲页，如果这些页是dirty的，则在转换为空闲页之前，先要把页的内容换出（也称为launder, 洗净）到swap分区对应的swap page中。

扫描页表是一项艰巨的任务，因为除了内核空间，用户地址空间有将近 3G 的空间，真正的程序很少能够用这么多。因此，充分利用虚存管理能够提升扫描页表的速度。

页面置换具体流程

现在我们来介绍以下 kswap_main 是如何一步一步完成 swap 的操作的。正如前面介绍过的，swap 需要完成3件事情，下面对应的是这三个操作的具体细节：

断开足够多的页表项PTE

kswapd_main函数通过循环调用函数swap_out_mm并进一步调用swap_out_vma，来查找ucore中所有存在的虚存空间，并总共断开 m 个 PTE 到物理页的映射。为何是查找ucore中所有存在的虚存空间,而不是直接扫描每个进程的虚存空间呢？这是因为虽然每个用户进程都拥有一个自己的虚存空间，但还存在虚存空间在多个进程之间的共享的情况。所以遍历虚存空间而不是遍历每个进程可避免某个虚存空间被很多进程共享进而被 kswapd 过度压榨所带来的不公平情况。可以想象，被过度压榨的虚存空间，同时又由于被很多进程共享从而有很高的概率被使用到，最终必然会导致频繁的内存访问错误异常#PF，给系统不必要的负担。

另外虽然 kswapd_main的第一个任务是断开 m 个 PTE 映射，但是实际上它对每个虚存空间都一次至多提出断开 32 个映射的需求，并循环遍历所有的虚存空间直到 m 得到满足。这样做的目的也是为了保证公平，使得每个虚存空间被交换出去的页的几率是近似相等的。Linux实际上应该有更好的实现，它根据虚存空间所实际使用的物理页的个数来决定断开的映射的

个数。）。这些断开的 PTE 映射所指向的物理页如果没有 PG_active 标记，则需要给它分配一个新的 swap entry，并做好标记，将 page 插入到 active_list 中去（同时也插入到 swap 的哈希表中），然后设置好相应的 page_ref 和 mem_map[offset] 的值。

当然，如果找不到空闲的 swap entry 可以分配（比如 swap 分区已经用光了），我们只能跳过这样的 PTE 映射，从下一个地址继续寻找出路；对于原来就已经标记了 PG_swap 的物理页，则只需要完成后面的工作，即调整引用计数就足够了。

断开的 PTE 被 swap_entry 取代，并取消 PTE_P 标记，，这样当出现#PF的时候，我们能够直接根据 PTE 上的值得到该页的数据实际是在swap 分区上的哪个位置上。

现在这一阶段的工作只是断开 PTE 映射，余下的工作后面会一步步完成。需要注意现在还不必要考虑一个 page 究竟是放在 active_list 还是放在 inactive_list 中，也还没涉及页换出操作。当 kswapd 断开足够数量的 PTE 映射以后，第一阶段的工作也就完成了。

当 kswapd 发现自己竭尽所能的遍历都无法满足断开 m 个链接的需求时，该怎么办？我们需要明确的是swap 操作的主要目的是释放物理页，而断开 PTE 映射是一个必要的步骤，作用是尽可能的扩大 inactive_list 中 page 的个数，为物理页的换出提供更大的基数（操作空间），但这并不是换页主要过程。所以为了防止在这一步陷入死循环，kswapd_main 最多会对全部虚存空间的链表尝试 rounds=16 次遍。

【注意】虚存管理数据结构mm_struct新增加了一个 swap_address ，表示上一次页换出操作结束时的地址。维护这个数据是避免每次 swap 操作都从虚存空间的起始地址开始，从而导致过多数量的重复且无效的遍历。

转换inactive page

refill_inactive_list 从函数名上可以看出，实际上就是遍历 active_list，把实际上不活跃的(inactive) page 从 active_list 上取下，放到 inactive_list 上，方便下一轮 page_launder 的操作。

页换出和释放页

通过 page_launder 函数，完成遍历 inactive_list并实现页的释放与换出。page_launder 的实现，涉及 ucore 内核代码设计的一个重要假设前提，ucore 的内核代码是不可抢占的（具体细节请参考4.5.4小节）。这部分和上述的refill_inactive_list 函数操作的先后顺序并不那么严格。通俗的解释就是 page_launder 实现的是把 inactive_list 中的 page 洗净（即完成 page 的释放和换出），当然也顺便实现了把实际上活跃的(active) 的 page 从 inactive_list 上取下，放回 active_list 的过程。

page_launder其实是比较复杂的过程，需要仔细的分析一下。page_launder 先检查一个 page 的 page_ref 是否 != 0。如果是，则表示该 page 实际上是 active 的，则把它移动到 active_list 上去；如果不是，则需要对该页进行释放和换出操作。具体过程如下（【注意】下

面讨论的是以 page_ref == 0 作为前提）

- 如果一个页的 mem_map 项为0，说明这个时候已经没有 PTE 映射指向它了，无论是物理页还是 swap 备份页。那么这个页也就没有必要洗净了。可以直接释放物理页以及相应的表示为swap entry的PTE 了。（同时还需处理 swap page 有关的链表，以下不再赘述）
- 如果一个页的 mem_map 项不为0，但是没有 PG_dirty 标记：page 数据结构里面有 PG_dirty 标记，swap 部分的代码根据这个标记来判断一个页是否需要被洗净（写到 swap 分区上）。这个 PG_dirty 标记在什么情况下设置，我们稍后会讨论。这种情况可以等价为物理页上的数据和 swap 分区上的数据是一致的，所以不需要洗净该页，因为该物理页本身就已经足够干净了。所以可以安全的和(1) 中的操作一样对该物理页进行释放。
- 如果一个页的确有 PG_dirty 标记：表示该页需要被洗净，这需要调用 swapfs_write 函数，完成将物理页写到磁盘上的操作。前面已经强调过，我们假设所有磁盘操作是异步IO。 先明确一下，当前的状态，page_ref=0 &&mem_map!=0 && PG_dirty，那么在执行写到磁盘的过程中就可能发生下面许多种可能的场景：

  a. swapfs_write 操作失败了。磁盘操作不像内存操作，它应该允许发生更多的错误。

  b. 其它进程又访问到相应的数据页，前面提到过，因为 PTE 的PTE_P标志位为0了，所以会产生#PF，内核会根据 PTE 的内容在 swap hash 里面查找到相应的物理页，并将它重新插入到相应的页表中，并更新该 page 的 page_ref 和 mem_map 的值。整个过程发生时，swapfs_write 还没有结束。那么当完成洗净一个页的操作时（写到swap 分区），swap 部分的代码应该有能力检测出这种变化。也就是在 swapfs_write 之后，需要再判断 page_ref 是否依然满足 inactive 的。

  c. 和b类似，不过不同的是，这次对物理页进行的是一个写操作。操作完成之后，进程又将该 PTE 指向的页释放掉了。那么当 swapfs_write 返回的时候，它面对的条件，可能就变成了 page_ref=0 &&mem_map=0 &&PG_dirty。它应该能够处理这个变化。

  d. 和c类似，不同的是，该page有两个不同的 PTE 映射。那么在swapfs_write 操作之前，状态可能是 page_ref =0&&mem_map=2&!PG_dirty，那么当c中的情况发生以后，该物理页的状态就可能变成了 page_ref=0&mem_map=1&PG_dirty了。swap 应该能够处理这种变化。 综上所述，page_launder 部分的代码变得相对复杂很多。大家可以参照程序了解ucore 是怎么解决这种冲突的。

其他注意事项

因为 swapfs_write 是异步操作，并且是对该 page 的操作，ucore 为了保证在操作的过程中，该页不被释放（比如一个进程通过#PF，增加 page_ref到1，然后又通过释放该page减少 page_ref到0，进而触发内核执行 page_free 的操作），分别在 swapfs_write 前后获得和释放该 page 的引用（page_ref_inc/page_ref_dec）。但事实证明，这种担心是多余的。理由很简单，当page_launder 操作一个页的时候，该页是被标记 PG_swap 的，这个标记一方面表

示 page 结构中的 index有意义，另一方面也说明这样的 page 的释放，只能够由 swap 部分的代码来完成（参见 pmm.c 以及后面 shmem.c 的处理）。所以，swap 在操作该 page 的时候，不可能有程序能够调用free_page释放该 page。

而相反的，mem_map 是一个需要保护的数据。此外，可以翻阅一下涉及到 page_ref 修改的 pmm 部分的代码，不难发现，当一个 page 从 PTE 断开的时候，也就是 page_ref 下降的时候，ucore会根据 PTE 上的硬件设置的 PTE_D 来设置 PG_dirty。其实这就足够了。因为 PG_dirty 并不需要时时刻刻都十分的准确，只要在 swap 尝试判断该 page 是否需要洗净的时候，PG_dirty 是正确的，就足够了。所以只需要保证每次 page_ref 下降的时候，PG_dirty 是正确的即可。除此之外，在对每个页分配 swap_entry 的时候，需要保证标记 PG_dirty，因为毕竟是刚刚分配的，物理页的数据还从来没有写出去过。 总结一下，页换入换出的实现很复杂，但是相对独立。并且正是由于 ucore 的内核代码不可抢占使得实现变得相对容易一些。只要是不涉及 IO 操作，大部分过程都可以认为处于不可抢占的内核执行过程。

创建并执行用户线程

实验目标

到proj12为止，ucore还一直没有用户线程。而用户线程与用户进程的区别在于操作系统用户进程管理除了涉及与执行过程相关的调度、进程上下文切换、执行状态变化外，还需管理内存、文件等资源，而用户线程只管理与执行过程相关的调度、上下文切换、执行状态变化。这样使得在执行线程创建、删除和线程上下文切换时的开销比对进程做类似的事情要少很大的开销。从而在操作系统的用户线程管理下，可以让应用程序开发员开发多线程应用软件的执行效率更高。

为了支持用户线程，我们还需对现有的进程管理进行有限扩展，在了解线程基本原理的情况下，设计并实现ucore对用户线程的基本支持。

概述

实现描述

proj12是lab3的最后一个project。它在proj10.1（中间还有proj10.2/10.3/10.4/11）的基础上实现了对用户线程的支持，主要参考了Linux的线程实现思路，把线程作为一个共享内存等资源的轻量级进程看待，扩展设计了进程控制块中支持用户线程的成员变量和与进程管理相关的系统调用，使得在现有进程管理的基础上，做相对较小的改动，就支持线程模型了。

项目组成

```
proj12
├── …
|   ├── process
|   |   ├── proc.c
|   |   ├── proc.h
|   |   └── …
|   ├── syscall
|   |   ├── syscall.c
|   |   └── …
└── user
    ├── libs
    |   ├── clone.S
    |   ├── lock.h
    |   ├── thread.c
    |   ├── thread.h
    |   ├── ulib.c
    |   └── ulib.h
    ├── threadfork.c
    ├── threadtest.c
    ├── threadwork.c
    └── …
```

相对于proj11，主要增加和扩展的文件如下：

- kern/proc.[ch]：扩展进程控制块，增加线程组成员变量，并扩展和增加与线程管理相关的函数；
- kern/syscall.c：增加用于线程创建的sys_clone系统调用；
- user/libs/*.[chS]：实现用户态创建线程的库调用函数、访问系统调用函数；
- user/thread*.c：线程支持用户测试用例。

编译运行

首先确保proj12的kern/proc.c中的user_main函数中的代码为：

```
static intuser_main(void *arg) {#ifdef TEST    KERNEL_EXECVE2(TEST, TESTSTART, TESTSIZE);#else    KERNEL_EXECVE(threadtest);#endif    panic("user_main execve failed.\n");}
```

编译并运行proj12的命令如下：

```
make
make qemu
```

则可以得到如下显示界面：

```
thuos:~/oscourse/ucore/i386/lab3_process/proj12$ make qemu
(THU.CST) os is loading ...
……
++ setup timer interrupts
kernel_execve: pid = 3, name = "threadtest".
thread ok.
child ok.
threadtest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:454:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

这其实是ucore首先创建了一个用户进程usermain，然后此用户进程通过调用sys_execv执行的是users/threadtest.c中的代码：

```
#include <ulib.h>
#include <stdio.h>
#include <thread.h>
inttest(void *arg) {
    cprintf("child ok.\n");
    return 0xbee;
}
intmain(void) {
    thread_t tid;
    assert(thread(test, NULL, &tid) == 0);
    cprintf("thread ok.\n");
    int exit_code;
    assert(thread_wait(&tid, &exit_code) == 0 && exit_code == 0xbee);
    cprintf("threadtest pass.\n");
    return 0;
}
```

usermain用户进程调用了thread用户库函数来创建了一个用户线程test，这个用户线程同享了usermain用户进程的地址空间，然后usermain用户进程就调用thread_wait函数等待用户线程结束。用户线程test在结束执行时，会设置退出码为0xbee。用户进程usermain会检查此退出码，看用户线程是否结束。上述执行过程其实包含了对用户线程整个生命周期的管理。下面我们将从原理和实现两个方面对此进行进一步阐述。

【原理】线程的属性与特征分析

线程概念的提出是计算机系统的技术发展和操作系统对进程管理优化过程的自然产物。随着计算机处理能力的提高，内存容量的加大，在一个计算机系统中会存在大量的进程，为了提高整个系统的执行效率，需要进程间能够进行简洁高效的数据共享和进程切换，进程创建和进程退出。这样就会产生一个自然的想法：能否改进进程模型，提供一个简单的数据共享机制，能否加快进程管理（特别是进程切换）的速度？再仔细看看进程管理的核心数据结构进程控制块和处理的流程，就可以发现，在某种情况下，进程的地址空间隔离不一定是一个必须的需求。假定进程间是可以相互“信任”的（即进程间是可信的），那么我们就不必需要地址空间隔离，而是让这些相互信任的进程共用（共享）一个地址空间。这样一下子就解决上述两个问题：在一个地址空间内，一个进程对某内存单元的修改（即写内存单元操作）可以让其他进程马上看见（即读内存单元操作），这是共享地址空间带来的天然好处；另外由于进程共享了地址空间，所以在创建进程或回收进程的时候，只要不是相互信任的进程组的第一个或最后一个，就没必要再创建一个地址空间或回收地址空间，节省了创建进程和退出进程的执行开销；而且对于频繁发生的进程切换操作而言，由于不需要切换页表，所以TLB中缓存的虚拟地址—物理地址映射关系（页表项的缓存）不必清除或失效，减少了进程切换的开销。

为了区别已有的进程概念，我们把这些共享资源的进程称为线程。从而把进程的概念进行了细化。原有的进程概念缩小为资源管理的最小单位，而线程是指令执行控制流的最小单位。且线程属于进程，是进程的一部分。一个进程至少需要一个线程作为它的指令执行单元，进程管理主要是资源（如内存空间等）分配、使用和回收的管理，而线程管理主要是指令执行过程和切换过程的管理。一个进程可以拥有多个线程，而这些线程共享其所属进程所拥有的资源。这样，采用多线程模型来设计应用程序，可以使得程序的执行效率也更高。

从执行线程调度时CPU所处特权级角度看，有内核级线程模型和用户级线程模型两种线程模型。内核级线程模型由操作系统在核心态进行调度，每个线程有对应的进程控制块结构。用户级线程模型有用户态的线程管理库在用户态进行调度，操作系统不能“感知”到这样的线程存在，所以也就没有对应进程控制块来描述它。相对而言，由于用户级线程模型在执行线程调度切换时不需从用户态转入核心态，开销相对较小，而内核级线程模型虽然开销相对较大，但由于操作系统直接负责管理，所以在执行的灵活性上由于用户级线程模型。比如用户级线程模型中某线程执行系统调用而可能被操作系统阻塞，这会引起同属于一个进程的其他线程都被阻塞。但内核级线程模型不会有这种情况后发生。这两种线程模型还可以结合在一起形成一种“混合”线程模型。“混合”线程模型通常都能带来更高的效率，但也带来更大的实现难度和实现代价。ucore出于“简单”的设计思路，参考Linux实现了内核级线程模型。

从线程执行时CPU所处特权级角度看，有内核线程和用户线程之分，内核线程共享操作系统内核运行过程中的所有资源，最主要的就是内核虚拟地址空间，但没有内核线程有各自独立的核心栈，且没有用户态的地址空间。用户线程共享属于同一用户进程的所有资源，最主要

的就是用户虚拟地址空间，所以同享同一用户进程的进程控制块所描述的一个页表和一个内存管理数据结构。接下来我们看看ucore中如何具体实现用户线程这个概念。

【实现】创建并执行用户线程

数据结构扩展：进程控制块和用户线程数据结构

由于ucore采用的是内核级线程模型，所以一个用户线程有一个进程控制块，但由于用户线程不同于用户进程，所以要对已有进程控制块进行简单扩展，这个扩展其实就是要表达共享了同一进程的线程组的关系：

```
struct proc_struct {
// the threads list including this proc which share resource
list_entry_t thread_group;
};
```

这样便于查找属于同一用户进程的所有用户线程。另外，属于同一用户进程的用户线程能够共享的进程控制块的主要成员变量包括：

```
struct mm_struct *mm;  // Process's memory management field
uintptr_t cr3; // CR3 register: the base addr of Page Directroy Table(PDT)
```

这样确保了属于同一用户进程的用户线程能共享同一用户地址空间。在对于其他一些与线程执行相关的成员变量，比如state、kstack、tf等，都有各自独立的数据存在。由于属于同一用户进程的不同用户线程除了在内核地址空间需要有不同的内核栈空间外，在用户地址空间也需要有不同的用户栈空间。为此，我们还需有一个数据结构来描述。ucore把这个数据机构放在用户态函数库中（user/libs/thread.h）：

```
typedef struct {
    int pid;
    void *stack;
} thread_t;
```

thread_t是对进程控制块的补充，pid是此线程的id标识（与进程控制块的pid一致），stack是用户栈的起始地址。通过这两方面的扩展，在数据结构层面就已经做好对用户线程的支持了。

创建用户线程

创建用户线程的执行流程重用了创建用户进程的执行过程，但有一些微小的差别，下面我们逐一进行分析。用户态函数库提供了thread函数来给应用程序提供创建线程的接口。

```
int
thread(int (*fn)(void *), void *arg, thread_t *tidp) {
    if (fn == NULL || tidp == NULL) {
        return -E_INVAL;
    }
    int ret;
    uintptr_t stack = 0;
    if ((ret = mmap(&stack, THREAD_STACKSIZE, MMAP_WRITE | MMAP_STACK)) != 0) {
        return ret;
    }
    assert(stack != 0);

    if ((ret = clone(CLONE_VM | CLONE_THREAD, stack + THREAD_STACKSIZE, fn, arg)) < 0)
 {
        munmap(stack, THREAD_STACKSIZE);
        return ret;
    }

    tidp->pid = ret;
    tidp->stack = (void *)stack;
    return 0;
}
```

从中可以看出，thread函数首先需要给用户线程分配用户栈，这里采用的是mmap函数（最终访问sys_mmap系统调用）来完成栈空间分配。其实ucore并没有真的给用户线程分配栈空间，这里采用了Demanding Paging（按需分页）技术来减少分配栈空间的时间和空间开销。另外，还调用了clone函数来完成用户线程的创建，clone进一步调用了sys_clone系统调用接口来要求ucore完成创建线程的服务。sys_clone与创建进程的sys_fork系统调用接口有何区别？它们的区别是创建标志位clone_flags：

- 调用clone创建线程的创建标志位：CLONE_VM | CLONE_THREAD
- 调用fork创建进程的创建标志位 ：0

【注意】另外clone函数要完成的具体工作是放在/usr/libs/clone.S中的函数**clone**来实现的。为何clone函数的内容不直接放到clone函数中用C语言来实现呢？这里其实只能用汇编来实现。因为对于用户进程而言，采用fork函数创建用户进程时，对于进程用户空间的设置采用的是整体复制或基于COW机制的整体共享方式，这样就可保证不同进程的堆和栈最终位于不同的用户地址空间。而对于用户线程而言，它需要共享父进程的地址空间，但又需要有自己的栈空间（此空间虽然父进程也可以“看到”，但不能用，只能归线程自己专用），所以在访问sys_clone系统调用前，用户线程用的是父进程的用户堆栈，在此系统调用返回后，用户线程已经被创建并开始在用户态执行，此时sp已经指向了新进程建立的新用户栈（见usr/libs/thread.c），而不是父进程的用户栈，这也就意味着在从系统调用返回后，子进程已经无法读位于父进程栈上的局部变量等数据。这样新线程需要使用寄存器（其实全局变量也行，只要避免使用位于进程栈上的局部变量即可）来进行数据处理，完成用户线程的继续执行。为了能够精确控制这个执行过程，不得不采用汇编来写，如果用C语言来写，你无法确保编译器不创建或使用局部变量来完成处理过程。

在内核中，创建线程的服务和创建进程的服务都是通过do_fork来实现的，而这个创建标志位clone_flags就决定了如何创建线程，查看do_fork函数：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    ……
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
     ……
    {
          ……
        if (clone_flags & CLONE_THREAD) {
            list_add_before(&(current->thread_group), &(proc->thread_group));
        }
    }
 ……
}
```

如果进一步分析copy_mm函数，如果clone_flags有CLONE_VM位，则执行：

```
……
    mm_count_inc(mm);
    proc->mm = mm;
    proc->cr3 = PADDR(mm->pgdir);
……
```

后两条语句说明了被创建的新线程重用了当前进程控制块的mm成员变量，并重用同样的页表。而如果clone_flags有CLONE_THREAD位，则会把被创建的新线程的进程控制块成员变量thread_group链接到当前进程控制块的牵头的链表thread_group中在，这样所有当前进程创建的线程都会在当前进程牵头的链表thread_group中找到。而在其他操作中，创建线程与创建进程的处理逻辑是一样的。

退出用户线程

在退出线程方面，与进程退出相比，区别不大。只是在执行do_exit的时候，调用de_thread(current)函数，进一步把自身从线程组链表中删除，其他方面都与进程执行do_exit一致；而父进程执行用户库函数thread_wait来通过sys_wait系统调用接口进一步调用内核函数do_exit完成对退出的子进程资源的最后回收。与进程的wait函数相比，主要的区别是：用户库函数thread_wait通过munmap函数完成了对线程用户栈空间的回收。与proj12以前实现的do_exit函数相比，主要的区别是：在查找是否有子线程的执行状态处于PROC_ZOMBIE时，除了搜索此进程的每个子进程外，还搜索此进程所属线程组的每个线程的所有子进程。其他方面的处理都是大致相同的。

进程运行状态转变过程

分析完从进程/线程从创建到退出的整个过程，我们需要在从全局的角度来看看进程/线程在做整个运行过程中的运行状态转变过程。在执行状态转变过程中，ucore在调度过程总，并没有区分线程和进程，所以进程和线程的执行状态转变是一致的，分析的结果适合用户线程和用户进程的执行过程。

首先为了描述进程/线程的整个状态集合，ucore在kern/process/proc.h中定义了进程/线程的运行状态：

```
// process's state in his life cycle
enum proc_state {
    PROC_UNINIT = 0,  // uninitialized
    PROC_SLEEPING,    // sleeping
    PROC_RUNNABLE,    // runnable(maybe running)
    PROC_ZOMBIE,      // almost dead, and wait parent proc to reclaim his resource
};
```

这与操作系统原理讲解的五进程执行状态相比，少了一个PROC_RUNNING态（表示正在占用CPU执行），这是由于在ucore中，用current（基于proc_strcut数据结构）进程控制块指针指向了当前正在运行的进程/线程PROC_RUNNING态，所以就没必要再增加一个PROC_RUNNING态了。那么那些事件或内核函数会触发状态的转变呢？通过分析uore源码，我们可以得到如下表示：

进程状态	转变原因
PROC_UNINIT	执行 alloc_proc 函数
PROC_SLEEPING	执行 try_free_pages, do_wait,do_sleep 函数
PROC_RUNNABLE	执行 proc_init, wakeup_proc 函数，有此状态的进程可处于就绪准备状态或占用 CPU 运行状态
PROC_ZOMBIE	执行 do_exit 函数

当父进程得到子进程的通知，回收完子进程控制块所占内存后，这个进程就彻底消失了。我们也可以用一个类似有限状态自动机来表示状态的变化：（需要用visio重画）

```
process state changing:

  alloc_proc                                 RUNNING
      +                                   +--<----<--+
      +                                   + proc_run +
      V                                   +-->---->--+
PROC_UNINIT -- proc_init/wakeup_proc --> PROC_RUNNABLE -- try_free_pages/do_wait/do_sl
eep --> PROC_SLEEPING --
                                           A      +
                        +
                                           |      +--- do_exit --> PROC_ZOMBIE
                        +
                                           +
                        +
                                           -----------------------wakeup_proc---------
-------------------------
```

进程调度

实验目标

在只有一个或几个CPU的计算机系统中，进程数量一般远大于CPU数量，CPU是一个稀缺资源，多个进程不得不分时占用CPU执行各自的工作，操作系统必须提供一种手段来保证各个进程“和谐”地共享CPU。为此，需要在lab3的基础上设计实现ucore进程调度类框架以便于设计各种进程调度算法，同时以提高计算机整体效率和尽量保证各个进程“公平”地执行作为目标来设计各种进程调度算法。

proj13/13.1/13.2概述

实现描述

project13是lab4的第一个项目，它基于lab3的最后一个项目proj12。主要参考了Linux-2.6.23设计了一个简化的进程调度类框架，能够在此框架下实现不同的调度算法，并在此框架下实现了一个简单的FIFO调度算法。接下来的proj13.1在此调度框架下实现了轮转（Round Robin，简称RR）调度算法，proj13.2在此调度框架下实现了多级反馈队列（Multi-Level Feed Back，简称MLFB）调度算法。

项目组成

```
proj13
├── kern
│   ├── ......
│   ├── process
│   │   ├──……
│   │   ├── proc.h
│   │   └── proc.c
│   ├── schedule
│   │   ├── sched.c
│   │   ├── sched_FCFS.c
│   │   ├── sched_FCFS.h
│   │   └── sched.h
│   └── trap
│       ├── trap.c
│       └── ……
└── user
    ├── matrix.c
    └── ……

17 directories, 129 files
```

相对与proj12，proj13增加了3个文件，修改了相对重要的5个文件。主要修改和增加的文件如下：

- process/proc.[ch]：扩展了进程控制块的定义能够把处于就绪态的进程放入到一个就绪队列中，并在创建进程时对新扩展的成员变量进行初始化。
- schedule/sched.[ch]：增加进程调度类的定义和相关进程调度类的共性函数。
- schedule/sched_FCFS.[ch]：基于进程调度类的FCFS调度算法实例的设计实现；
- schedule/sched.[ch]：实现了一个先来先服务（First Come First Serve）策略的进程调度。
- user/matrix.c：矩阵乘用户测试程序

编译运行

编译并运行proj13的命令如下：

```
make
make qemu
```

则可以得到如下显示界面

```
(THU.CST) os is loading ...

Special kernel symbols:
……
++ setup timer interrupts
kernel_execve: pid = 3, name = "matrix".
fork ok.
pid 4 is running (1000 times)!.
pid 4 done!.
pid 5 is running (1400 times)!.
pid 5 done!.
……
pid 22 is running (33400 times)!.
pid 22 done!.
pid 23 is running (33400 times)!.
pid 23 done!.
matrix pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:456:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

这其实是在采用简单的FCFS调度方法来执行matrix用户进程，这个matrix进程将创建20个进程来各自执行二维矩阵乘的工作。Ucore将按照FCFS的调度方法，一个一个地按创建顺序执行每个子进程。一个子进程结束后，再调度另外一个子进程运行。这实际上看不出ucore是如何具体实现进程调度类框架和FCFS调度算法的。下面我们首先介绍一下进程调度的基本原理，然后再分析ucore的调度框架和调度算法的实现。

【原理】进程调度

需要进程调度的理由很简单，即充分利用计算机系统中的CPU资源，让计算机系统能够多快好省地完成我们让它做的各种任务。为此，可在内存中可存放数目远大于计算机系统内CPU个数的进程，让这些进程在操作系统的进程调度器的调度下，能够让进程高效（高的吞吐量--throughput）、及时（低延迟--latency）、公平（fairness）地使用CPU。为此调度器可设计不同的调度算法来选择进程，这体现了进程调度的策略，同时还需并进一步通过进程的上下文切换（context switch）来完成进程切换，这体现了进程调度的机制。总体上说，我们需要何时调度（调度的时机）、是否能够在内核执行的任意位置进行调度（调度的方式）、如果完成进程切换（上下文切换）、如果选择“合适”的进程执行（调度策略/调度算法）、如果评价选择的合理性（进程调度的指标）。了解上述细节，也就可以说是了解了进程调度。

进程调度的指标

不同的进程调度算法具有不同的特征，为此需要建立衡量一个算法的基本指标。一般而言，衡量和比较各种进程调度算法性能的主要因素如下所示：

- CPU利用率：CPU是计算机系统中的稀缺资源，所以应在有具体任务的情况下尽可能使CPU保持忙，从而使得CPU资源利用率最高。
- 吞吐量：CPU运行时的工作量大小是以每单位时间所完成的进程数目来描述的，即称为吞吐量。
- 周转时间：指从进程创建到作进程结束所经过的时间，这期间包括了由于各种因素（比如等待I/O操作完成）导致的进程阻塞，处于就绪态并在就绪队列中排队，在处理机上运行所花时间的总和。
- 等待时间：即进程在就绪队列中等待所花的时间总和。因此衡量一个调度算法的简单方法就是统计进程在就绪队列上的等待时间。
- 响应时间：指从事件（比如产生了一次时钟中断事件）产生到进程或系统作出响应所经过的时间。在交互式桌面计算机系统中，用户希望响应时间越快越好，但这常常要以牺牲吞吐量为代价。

  这些指标其实是相互有冲突的，响应时间短也就意味着在相关事件产生后，操作系统需要迅速进行进程切换，让对应的进程尽快响应产生的事件，从而导致进程调度与切换的开销增大，这会降低系统的吞吐量。

进程调度的时机

进程调度发生的时机（也称为调度点）与进程的状态变化有直接的关系。回顾进程状态变化图，我们可以看到进程调度的时机直接与进程在运行态<-->退出态/就绪态/阻塞态的转变时机相关。简而言之，引起进程调度的时机可归结为以下几类：

- 正在执行的进程执行完毕，需要选择新的就绪进程执行。
- 正在执行的进程调用相关系统调用（包括与I/O操作，同步互斥操作等相关的系统调用）导致需等待某事件发生或等待资源可用，从而将自己阻塞起来进入阻塞状态。
- 正在执行的进程主动调用放弃CPU的系统调用，导致自己的状态为就绪态，且把自己重新放到就绪队列中。
- 等待事件发生或资源可用的进程等待队列，从而导致进程从阻塞态回到就绪态，并可参与到调度中。
- 正在执行的进程的时间片已经用完，致自己的状态为就绪态，且把自己重新放到就绪队列中。
- 在执行完系统调用后准备返回用户进程前的时刻，可调度选择一新用户进程执行
- 就绪队列中某进程的优先级变得高于当前执行进程的优先级，从而也将引发进程调度。

进程调度的方式

这里需要注意，存在两种进程抢占处理器的调度方式：

- 可抢占式（可剥夺式，preemptive)：就绪队列中一旦有某进程的优先级高于当前正在执行的进程的优先级时，操作系统便立即进行进程调度，完成进程切换。
- 不可抢占式（不可剥夺式non_preemptive）：即使在就绪队列存在有某进程优先级高于当前正在执行的进程的优先级时，当前进程仍将占用处理机执行，直到该进程自己进入阻塞状态，或时间片用完，或在执行完系统调用后准备返回用户进程前的时刻，才重新发生调度让出处理机。

显然，可抢占式调度可有效减少等待时间和响应时间，但会带来较大的其他管理开销，使得吞吐量等的性能指标比不可抢占式调度要低。所以一般在桌面计算机中都支持可抢占式调度，使得用户可以得到更好的人机交互体验，而在服务器领域不必非要可抢占式调度，而通常会采用不可抢占式调度，从而可提高系统的整体吞吐量。

进程调度的策略/算法

在早期操作系统的调度方式大多数是非剥夺的，这是由于早期的应用一般是科学计算或事务处理，不太把人机交互的响应时间指标放在首要位置。在这种情况下，正在运行的进程可一直占用CPU直到进程阻塞或终止。这种方式的调度算法可以很简单，且比较适用对于响应时间不关心或者关心甚少的批处理科学计算或事务处理应用。随着计算机的应用领域进一步扩

展，计算机更多地用在了多媒体等人机交互应用上，为此采用可抢占式的调度方式可在一个进程终止或阻塞之前就剥夺其执行权，把CPU尽快分配给另外的“更重要”进程，使得就绪队列中的进程有机会响应它们用户的IO事件。基于这两种方式的调度算法如下：

- 先来先服务（FCFS）调度算法：处于就绪态的进程按先后顺序链入到就绪队列中，而FCFS调度算法按就绪进程进入就绪队列的先后次序选择当前最先进入就绪队列的进程来执行，直到此进程阻塞或结束，才进行下一次的进程选择调度。FCFS调度算法采用的是不可抢占的调度方式，一旦一个进程占有处理机，就一直运行下去，直到该进程完成其工作，或因等待某一事件而不能继续执行时，才释放处理机。操作系统如果采用这种进程调度方式，则一个运行时间长且正在运行的进程会使很多晚到的且运行时间短的进程的等待时间过长。
- 短作业优先（SJF）调度算法：其实目前作业的提法越来越少，我们姑且把“作业”用“进程”来替换，改称为短进程优先调度算法，此算法选择就绪队列中确切（或估计）运行时间最短的进程进入执行。它既可采用可抢占调度方式，也可采用不可抢占调度方式。可抢占的短进程优先调度算法通常也叫做最短剩余时间优先（Shortest Remaining Time First，SRTF）调度算法。短进程优先调度算法能有效地缩短进程的平均周转时间，提高系统的吞吐量，但不利于长进程的运行。而且如果进程的运行时间是“估计”出来的话，会导致由于估计的运行时间不一定准确，而不能实际做到短作业优先。
- 时间片轮转（RR）调度算法：RR 调度算法与FCFS 调度算法在选择进程上类似，但在调度的时机选择上不同。RR调度算法定义了一个的时间单元，称为时间片（或时间量）。一个时间片通常在1～100 ms之间。当正在运行的进程用完了时间片后，即使此进程还要运行，操作系统也不让它继续运行，而是从就绪队列依次选择下一个处于就绪态的进程执行，而被剥夺CPU使用的进程返回到就绪队列的末尾，等待再次被调度。时间片的大小可调整，如果时间片大到让一个进程足以完成其全部工作，这种算法就退化为FCFS调度算法；若时间片设置得很小，那么处理机在进程之间的进程上下文切换工作过于频繁，使得真正用于运行用户程序的时间减少。时间片可以静态设置好，也可根据系统当前负载状况和运行情况动态调整，时间片大小的动态调整需要考虑就绪态进程个数、进程上下文切换开销、系统吞吐量、系统响应时间等多方面因素。
- 高响应比优先（Highest Response Ratio First，HRRF）调度算法：HRRF调度算法是介于先来先服务算法与最短进程优先算法之间的一种折中算法。先来先服务算法只考虑进程的等待时间而忽视了进程的执行时间，而最短进程优先调度算法只考虑用户估计的进程的执行时间而忽视了就绪进程的等待时间。HRRF调度算法二者兼顾，既考虑进程等待时间，又考虑进程的执行时间，为此定义了响应比（Rp）这个指标：

```
Rp=（等待时间+预计执行时间）/执行时间=响应时间/执行时间
```

上个表达式假设等待时间与预计执行时间之和等于响应时间。HRRF调度算法将选择Rp最大值的进程执行，这样既照顾了短进程又不使长进程的等待时间过长，改进了调度性能。但HRRF调度算法需要每次计算各各个进程的响应比Rp，这会带来较大的时间开销（特别是在就绪进程个数多的情况下）。

- 多级反馈队列（Multi-Level Feedback Queue）调度算法：在采用多级反馈队列调度算法的执行逻辑流程如下：

    - 设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二队次之，其余队列优先级依次降低。仅当第1～i-1个队列均为空时，操作系统调度器才会调度第i个队列中的进程运行。赋予各个队列中进程执行时间片的大小也各不相同。在优先级越高的队列中，每个进程的执行时间片就越小或越大（Linux-2.4内核就是采用这种方式）。
    - 当一个就绪进程需要链入就绪队列时，操作系统首先将它放入第一队列的末尾，按FCFS的原则排队等待调度。若轮到该进程执行且在一个时间片结束时尚未完成，则操作系统调度器便将该进程转入第二队列的末尾，再同样按先来先服务原则等待调度执行。如此下去，当一个长进程从第一队列降到最后一个队列后，在最后一个队列中，可使用FCFS或RR调度算法来运行处于此队列中的进程。
    - 如果处理机正在第i（i>1）队列中为某进程服务时，又有新进程进入第k（k<i）的队列，则新进程将抢占正在运行进程的处理机，即由调度程序把正在执行进程放回第i队列末尾，重新将处理机分配给处于第k队列的新进程。

  从MLFQ调度算法可以看出长进程无法长期占用处理机，且系统的响应时间会缩短，吞吐量也不错（前提是没有频繁的短进程）。所以MLFQ调度算法是一种合适不同类型应用特征的综合进程调度算法。

- 最高优先级优先调度算法：进程的优先级用于表示进程的重要性及运行的优先性。一个进程的优先级可分为两种：静态优先级和动态优先级。静态优先级是在创建进程时确定的。一旦确定后，在整个进程运行期间不再改变。静态优先级一般由用户依据包括进程的类型、进程所使用的资源、进程的估计运行时间等因素来设置。一般而言，若进程需要的资源越多、估计运行的时间越长，则进程的优先级越低；反之，对于I/O bounded的进程可以把优先级设置得高。动态优先级是指在进程运行过程中，根据进程执行情况的变化来调整优先级。动态优先级一般根据进程占有CPU时间的长短、进程等待CPU时间的长短等因素确定。占有处理机的时间越长，则优先级越低，等待时间越长，优先级越高。那么进程调度器将根据静态优先级和动态优先级的总和现在优先级最高的就绪进程执行。

操作系统中为了能够让每个进程都有机会运行，需要给每个进程分配一个时间片，当一个进程的时间片用完以后，操作系统的调度器就会让当前进程放弃CPU，而选择另外一个进程占用CPU执行。为了有效地支持进程调度所需的时间片，ucore设计并实现了一个timer（计时器）功能。这样，通过timer就对基于时间事件的调度机制提供了基本支持。

【实现】进程调度

内核的抢占性

调度本质上体现了对CPU资源的抢占。对于用户进程而言，由于有中断的产生，可以随时打断用户进程的执行，转到操作系统内部，从而给了操作系统以调度控制权，让操作系统可以根据具体情况（比如用户进程时间片已经用完了）选择其他用户进程执行。这体现了用户进程的可抢占性（preemptive）。但如果把ucore操作系统也看成是一个特殊的内核进程或多个内核线程的集合，那ucore是否也是可抢占的呢？其实ucore 内核执行是不可抢占的（non-preemptive），即在执行“任意”内核代码时，CPU控制权可被强制剥夺。这里需要注意，不是在所有情况下ucore内核执行都是不可抢占的，有以下几种“固定”情况是例外：

1. 进行同步互斥操作，比如争抢一个信号量、锁（lab5中会详细分析）；
2. 进行磁盘读写等耗时的异步操作，由于等待完成的耗时太长，ucore会调用shcedule让其他就绪进程执行。

这几种情况其实都是由于当前进程所需的某个资源（也可称为事件）无法得到满足，无法继续执行下去，从而不得不主动放弃对CPU的控制权。如果参照用户进程任何位置都可被内核打断并放弃CPU控制权的情况，这些在内核中放弃CPU控制权的执行地点是“固定”而不是“任意”的，不能体现内核任意位置都可抢占性的特点。我们搜寻一下proj13的代码，可发现在如下几处地方调用了shedule函数：

调用进程调度函数schedule的位置和原因

编号	位置	原因
1	swap.c::try_free_page	用户进程申请内存无法得到满足，主动放弃对 CPU 控制权，等待 kswapd 内核线程释放出足够多的空闲空间。
2	proc.c::do_exit	用户线程执行结束，主动放弃 CPU 控制权。
3	proc.c::do_wait	用户线程等待子进程结束，主动放弃 CPU 控制权。
4	proc.c::do_sleep	用户线程要睡眠一段时间，主动放弃 CPU 控制权。
5	proc.c::init_main	1. initproc 内核线程等待所有用户进程结束，如果没有结束，就主动放弃 CPU 控制权; 2. initproc 内核线程在所有用户进程结束后，让 kswapd 内核线程执行 10 次，用于回收空闲内存资源
6	proc.c::cpu_idle	idleproc 内核线程的工作就是等待有处于就绪态的进程或线程，如果有就调用 schedule 函数
7	sync.h::lock	在获取锁的过程中，如果无法得到锁，则主动放弃 CPU 控制权
8	trap.c::trap	如果在当前进程在用户态被打断去，且当前进程控制块的成员变量 need_resched 设置为 1，则当前线程会放弃 CPU 控制权

仔细分析上述位置，第1、2、3、4、7处的执行位置体现了由于访问磁盘数据的资源或获取锁资源一时等不到满足、进程要退出、进程要睡眠等原因而不得不主动放弃CPU。第5、6处的执行位置比较特殊，initproc内核线程等待用户进程结束和执行kwapd内核线程而执行schedule函数；idle内核线程在没有进程处于就绪态时才执行，一旦有了就绪态的进程，它将执行schedule函数完成进程调度。这里只有第8处的位置比较特殊：

```
if (!in_kernel) {
     ……
    if (current->need_resched) {
         schedule();
     }
 }
```

这里表明了只有当进程在用户态执行到"任意"某处用户代码位置时发生了中断，且当前进程控制块成员变量need_resched为1（表示需要调度了）时，才会执行shedule函数。这实际上体现了对用户进程的可抢占性。如果没有第一行的if语句，那么就可以体现对内核代码的可抢占性。但如果要把这一行if语句去掉，我们不得不实现对ucore中的所有全局变量的互斥访问操作，以防止所谓的race condition现象（在lab5中有进一步讲述），这样ucore的实现复杂度会增加不少。

进程调度时机

我们可以知道ucore通过进程控制块的state成员变量来表示进程的运行状态，而且处于就绪态的进程和运行态的进程在ucore中的运行状态都是PROC_RUNNABLE。那如何进一步区分处于就绪态的进程和运行态的进程呢？在ucore中，所有处于就绪态的进程会链接在一个就绪队列rq（run queue的简称），而处于运行态的进程实际上市从就绪队列rq中被进程调度器选择出来的一个进程，此进程会从此就绪队列中断开，并开始占用CPU执行。这说明，没有挂在就绪队列且运行状态为PROC_RUNNABLE的进程是处于运行态的进程。

根据进程状态变化过程的分析，我们可以知道选择一个进程，把它从就绪态转变到运行态是进程调度器的主要工作。在ucore内核中哪些地方会进行调度呢？这其实与进程的运行状态变化的时机相关，回想4.4.5小节描述的进程状态变化过程，可以知道在如下一些地方是需要进行进程调度的。这里需要更深入地分析从运行态到其他状态相关转换的情况

运行态到就绪态的变化过程

首先来看运行态到就绪态的变化过程。进程主动/被动放弃CPU回到就绪态的起因有两个。一个起因是用户进程主动从运行态回到就绪态，即用户进程调用yield用户态库函数，此库函数接下来的调用路径如下：

```
yield(用户态)-->sys_yeild(用户态)-->sys_yeild（内核态）-->do_yield(内核态)
```

do_yeild内核函数仅仅把当前进程的进程控制块成员变量need_resched设置为1。

```
int
do_yield(void) {
    current->need_resched = 1;
    return 0;
}
```

另外一个起因是在进程调度器采用类似时间片轮转调度（RR）策略的前提下，用户进程由于时间片用完被动地从运行态回到就绪态的情况，即当时钟中断积累到一定的时间片段（给用户进程设置能够持续占用CPU运行的时间）后，如果一个用户进程还在持续运行，则ucore需要强制剥夺用户进程的CPU使用权，即把当前进程重新插入到就绪队列中，并从就绪队列中再选择另外一个“合适”的进程执行。时钟中断产生后，整个执行过程的函数调用路径如下：

```
vetcor32-->__alltraps-->trap-->trap_dispatch-->run_timer_list -->sched_class_proc_tick
-->XXX_proc_tick
```

注意XXX_proc_tick是某个具体进程调度器XXX对产生时钟中断后的特定处理函数。比如对于FCFS调度器而言，它没有考虑时间片轮转的情形，所以实际的FCFS_proc_tick函数啥也没干。但如果是RR调度器（在lab4/proj13.1中实现），这实际的RR_proc_tick则需要递减当前进程拥有的时间片，并判断当前进程的时间片是否已经用完，如果用完了，则需要把当前进程的进程控制块成员变量need_resched设置为1。

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

其实把当前进程的进程控制块成员变量need_resched设置为1只是指出了需要调度，但并没有实现调度。那需要在哪里完成调度呢？注意到上一小节“内核的抢占性”中，我们可以看到，实际上ucore只是在中断或异常返回的时候，如果在用户态被中断的当前用户进程need_resched设置为1，则会执行schedule函数，完成进程调度与切换。

运行态到就睡眠态的变化过程

参考上表（调用进程调度函数schedule的位置和原因），我们可以看到有有三个地方：第1、3、4处的执行代码会导致当前进程转变到睡眠态。第一个是由于当前进程申请内存无法得到满足，需通过执行try_free_pages函数，主动放弃对CPU控制权，等待kswapd内核线程释放出足够多的空闲空间。这里采用了等待队列的机制实现进程睡眠：

```
local_intr_save(intr_flag);
{
    wait_init(wait, current);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_KSWAPD;
    wait_queue_add(&kswapd_done, wait);
    if (kswapd->wait_state == WT_TIMER) {
        wakeup_proc(kswapd);
    }
}
local_intr_restore(intr_flag);
schedule();
```

上述代码很清楚地表明了当前进程的状态转变为睡眠状态，睡眠原因是WT_KSWAPD，即等待内核线程kswapd释放出更多的空闲内存，并执行schedule函数，把自身从就绪队列中删除，并选择新的就绪进程占用CPU执行。 第二个是do_wait函数。用户进程执行wait/waitpid用户库函数,并进一步调用sys_wait用户函数和sys_wait内核函数后，将调用do_wait函数完成实际的父进程等待子进程的工作。相关代码如下：

```
if (haskid) {
    current->state = PROC_SLEEPING;
    current->wait_state = WT_CHILD;
    schedule();
```

…… 此函数判断如果当前进程有子进程，则会设置当前进程状态转变为睡眠状态，睡眠原因是WT_CHILD，即等待子进程结束，并执行schedule函数，把自身从就绪队列中删除，并选择新的就绪进程占用CPU执行。 第三个是do_sleep函数。用户进程执行sleep用户库函数来实现n毫秒睡眠，这将进一步调用sys_sleep用户函数和sys_sleep内核函数后，最终调用do_sleep完成实际的睡眠操作。相关代码如下：

```
do_sleep(unsigned int time) {
    if (time == 0) {
        return 0;
    }
    bool intr_flag;
    local_intr_save(intr_flag);
    timer_t __timer, *timer = timer_init(&__timer, current, time);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_TIMER;
    add_timer(timer);
    local_intr_restore(intr_flag);

    schedule();
     ……
}
```

可以看出会设置当前进程状态转变为睡眠状态，睡眠原因是WT_TIMER，即等待定时器到时，并执行schedule函数，把自身从就绪队列中删除，并选择新的就绪进程占用CPU执行。这里采用了定时器机制（参见4.2.5小节），通过调用timer_init函数来创建定时器，并调用add_timer函数来设置定时器运转，当时间time到后，会唤醒当前进程。

运行态到就退出态的变化过程

当用户进程执行完毕（或者被要求强行退出）后，将执行do_exit函数完成对自身所占部分资源的回收，并执行进程调度切换。参考上表（调用进程调度函数schedule的位置和原因），我们可以看到第2处的do_exit函数的代码实现从运行态到退出态的变化过程，

```
current->state = PROC_ZOMBIE;
current->exit_code = error_code;
```

…… schedule();

可以看出会设置当前进程状态转变为退出状态，并执行schedule函数，把自身从就绪队列中删除，并选择新的就绪进程占用CPU执行。

进程切换过程

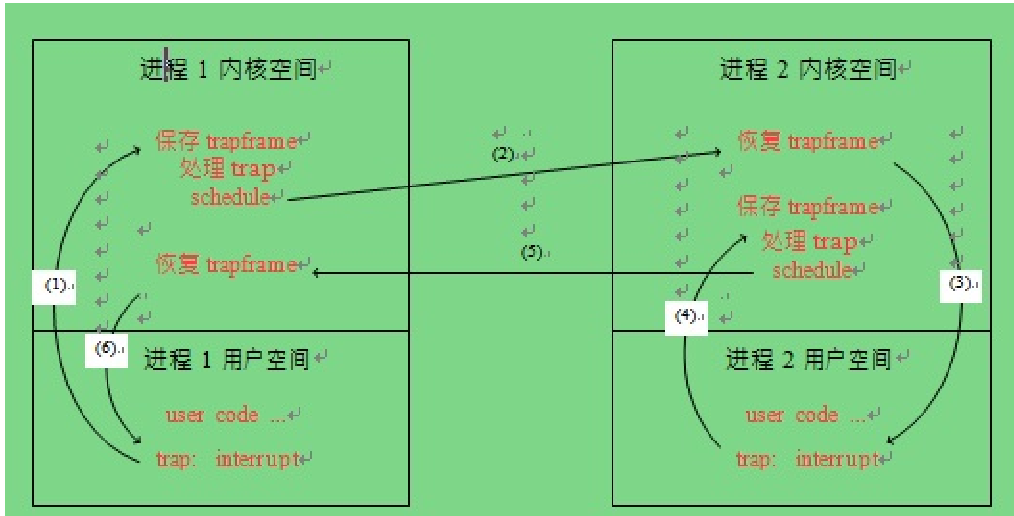
进程切换的具体过程发生在内核态，我们以对于基于时间片的进程调度过程来具体分析两个进程切换的过程。首先在执行进程 1 的用户代码时，出现了一个 trap (例如是一个 Timer中断)，这 个时候就会从进程 1 的用户态切换到内核态(过程(1))，并且保存好进程 1 的trapframe； 当ucore在处理中断时发现此进程的时间片已经用完，就会设置进程1的进程控制块的need_resched为1，表明需要进行进程调度，ucore 于是进一步执行 schedule函数，并通过此函数把进程1重新放回就绪队列，选择了另一个进程—进程2，并把进程2从就绪队列中摘除，这时需要调用proc_run函数来具体完成进程切换了。我们来看看切换的过程：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

从上面的代码可以看出，proc_run的函数参数proc是进程2，而current是进程1，首先调用load_esp0需要设置进程2用户态返回到内核态的内涵栈寄存器指针（即TS段的ESP0）；然后页表基址（CR3寄存器的内容）设置为进程2的页表基址，虽然换了一个页表基址，但由于所有进程在内核中都用的是同一个内核虚拟地址空间，所以切换后只要在内核态执行，并访问内核地址空间，那实际上没啥变化，而只是在进程2返回到用户态的时候，所用的用户地址空间与进程1的用户地址空间就不一样了；最后一步是调用switch_to函数切换进程运行相关的硬件寄存器内容（具体过程可回顾4.1.5节的第3小节“调度并执行内核线程initproc”）。一旦执行完switch_to的最后一个指令“RET”后，就彻底切换到进程2的执行环境了。

到了进程2的执行环境，首先继续执行进程 2 上一次在内核态的操作，并最终通过中断或异常的返回操作回到进程2的用户空间执行。类似上述描述过程，当进程 2 由于某种原因发生中断之后，并需要切换到进程 1；当再次切换到进程 1 时，会执行进程 1 上一次在内核调用

schedule (具体还要跟踪到 switch to 函数) 函数后的下一行代码，这行代码当然还是在进程1的上一次中断处理的位置处。最后当进程 1 的中断处理完毕的时候，执行权又会反交给进程 1 的用户代码。大致流程如下所示：



【问题】进程切换的工作可以在用户态实现吗？（提示，如果是用户线程，即执行环境不需要考虑页表和内核栈等，只在用户态执行，那就可以在用户态实现用户线程的切换，这就是通常用户态线程库的实现）

【问题】进程切换以后，当前进程是从哪里开始执行的？（提示，虽然还是同一个cpu，但是此时使用的资源已经完全不同了。）

【问题】内核在第一个用户进程运行的时候，需要进行哪些操作？（提示，内核运行第一个用户进程的过程，实际上是从启动时的内核状态切换到该程序的内核 状态的过程，而用户程序的起始状态的入口，即forkret等。）

进程调度类框架设计

设计思路

进程调度类框架的设计是为了更好地实行各种进程调度策略或算法，为此需要了解为了实行一个进程调度策略，到底需要实现哪些基本功能对应的数据结构？首先考虑到一个无论哪种调度算法都需要选择一个就绪进程来占用CPU运行。为此我们可把就绪进程组织起来，可用队列（双向链表）、二叉树、红黑树、数组…等不同的组织方式。

在操作方面，如果需要选择一个就绪进程，就可以从基于某种组织方式的就绪进程集合中选择出一个进程执行。需要注意，这里“选择”和“出”是两个操作，选择是在集合中挑选一个“合适”的进程，“出”意味着离开就绪进程集合。另外考虑到一个处于运行态的进程还会由于某种原因（比如时间片用完了）回到就绪态而不能继续占用CPU执行，这就会重新进入到就绪进程集合中。这两种情况就形成了调度器相关的三个基本操作：在就绪进程集合中选择、进入就绪进程集合和离开就绪进程集合。这三个操作属于调度器的基本操作。

在进程的执行过程中，就绪进程的等待时间和执行进程的执行时间是影响调度选择的重要因素，这两个因素随着时间的流逝和各种事件的发生在不停地变化，比如处于就绪态的进程等待调度的时间在增长，处于运行态的进程所消耗的时间片在减少等。这些进程状态变化的情况需要及时让进程调度器知道，便于选择更合适的进程执行。所以这种进程变化的情况就形成了调度器相关的一个变化感知操作： timer时间事件感知操作。这样在进程运行或等待的过程中，调度器可以调整进程控制块中与进程调度相关的属性值（比如消耗的时间片、进程优先级等），并可能导致对进程组织形式的调整（比如以时间片大小的顺序来重排双向链表等），并最终可能导致调选择新的进程占用CPU运行。这个操作属于调度器的进程调度属性调整操作。

数据结构和变量

在加上对进程调度相关所需的通用初始化操作，就形成了进程调度类框架的5个指针函数，每个具体的进程调度实例将分别实现这5个函数，完成不同的进程调度策略/算法。具体而言，在ucore中进程调度类定义如下：

```
struct sched_class {
    // the name of sched_class
    const char *name;
    // Init the run queue
    void (*init)(struct run_queue *rq);
    // put the proc into runqueue, and this function must be called with rq_lock
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    // get the proc out runqueue, and this function must be called with rq_lock
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    // choose the next runnable task
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    // dealer of the time-tick
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

这其实是对Linux内核的调度类的一种简化设计，需要注意这里的run_queue是就绪进程集合的一种组织方式，并不是特指队列（queue），我们根据调度算法的具体设计，也可采用Linux的CFS调度器的就绪进程队列组织结构--红黑树。在ucore中，不同调度算法需要设计不同的就绪进程队列组织结构。比如lab4/proj13实现的是FCFS调度算法，所以就绪进程只需能够按创建顺序链接如一个双向链表即可，为此设计的就绪进程队列组织结构就是一个双向链表：

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
};
```

在此结构中，还有一个proc_num成员变量，表示当前就绪进程的个数。对于lab4/proj13.1而言，它实现了RR调度算法，由于RR调度算法需要分析和调整每个进程的时间片，所以对就绪进程队列组织结构进行了小的扩展，增加了一个最大时间片的设定，用于比较当前进程的时间片是否已经超过了就绪进程队列设计的最大时间片范畴。其具体的设置如下：

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
};
```

对于lab4/proj13.2，实现了MLFQ调度算法，这里包含了4个不同级别的RR就绪队列，于是需要对就绪进程队列进行进一步扩展：

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
    list_entry_t rq_link;
};
```

其中增加的rq_link就用于把就绪进程放入到某个运行队列中。另外，对于RR调度算法和MLFQ调度算法而言，需要基于就绪进程和执行进程的时间片来调整就绪进程组织结构和抢占当前运行进程，所以需要对代表每个进程的进程控制块进行扩展，于是在lab4/proj13.1中对进程控制块进行了扩展，增加了time_slice成员变量来表示进程的时间片。

这样用于表示调度器操作的数据结构sched_class进程调度类和表示就绪进程组织形式的run_queue数据结构就绪进程队列就形成了进程调度类框架的主体。但如何在操作系统中运用这些数据结构，来实现与调度算法无关的调度框架呢？

调度点的关键调度相关函数

在本节中“进程调度时机”小节分析了进程在执行中状态变化过程的处理流程。虽然进程各种状态变化的原因和导致的调度处理各异，但其实仔细观察各个流程的共性部分，会发现其中只涉及了三个关键调度相关函数：wakup_proc、shedule、run_timer_list。如果我们能够让这三个调度相关函数的实现与具体调度算法无关，那么就可以认为ucore实现了一个与调度算法无关的调度框架。

wakeup_proc函数其实完成了把一个就绪进程放入到就绪进程队列中的工作，为此还调用了一个调度类接口函数sched_class_enqueue，这使得wakeup_proc的实现与具体调度算法无关。schedule函数完成了与调度框架和调度算法相关三件事情:把当前继续占用CPU执行的运行进程放放入到就绪进程队列中，从就绪进程队列中选择一个“合适”就绪进程，把这个“合适”的就绪进程从就绪进程队列中摘除。通过调用三个调度类接口函数

sched_class_enqueue、sched_class_pick_next、sched_class_enqueue来使得完成这三件事情与具体的调度算法无关。run_timer_list函数在每次timer中断处理过程中被调用，从而可用来调用调度算法所需的timer时间事件感知操作，调整相关进程的进程调度相关的属性值。通过调用调度类接口函数sched_class_proc_tick使得此操作与具体调度算法无关。

这里涉及了一系列调度类接口函数：

```
sched_class_enqueue
sched_class_dequeue
sched_class_pick_next
sched_class_proc_tick
```

这4个函数的实现其实就是调用某基于sched_class数据结构的特定调度算法实现的4个指针函数。采用这样的调度类框架后，如果我们需要实现一个新的调度算法，则我们需要定义一个针对此算法的调度类的实例，一个就绪进程队列的组织结构描述就行了，其他的事情都可交给调度类框架来完成。

进程调度策略/算法

FCFS调度算法的实现

FCFS调度算法不需要考虑执行进程的运行时间和就绪进程的等待时间，所以其实现很简单。首先其就绪进程队列是一个双向链表：

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
};
```

并在sched.c中声明了基于此数据结构的就绪进程队列rq。在执行wakup_proc时，把进程加入到就绪进程队列中（插入到rq的队列尾）。在执行schedule函数时，当前执行进程只会是已经退出或者已经睡眠两种情况，这时需要从就绪进程队列中选择最早进入就绪队列的进程（位于rq的队列头），然后把它从就绪队列中摘除。由于没有时间片的考虑，所以与run_timer_list函数相关的FCFS_proc_tick函数为空函数。我们在看看其他三个FCFS算法相关的函数实现。

FCFS_enqueue的函数实现如下：

```
static void
FCFS_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    proc->rq = rq;
    rq->proc_num ++;
}
```

即把一个就绪进程插入到就绪进程队列rq的队列尾，并把表示就绪进程个数的proc_num加一。

FCFS_pick_next的函数实现如下：

```
static struct proc_struct *
FCFS_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针。

FCFS_dequeue的函数实现如下：

```
static void
FCFS_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}
```

即把就绪进程队列rq的进程控制块指针的队列元素删除，并把表示就绪进程个数的proc_num减一。

RR调度算法的实现

RR调度算法的就绪队列在组织结构上也是一个双向链表，只是增加了一个成员变量，表明在此就绪进程队列中的最大执行时间片。而且在进程控制块proc_struct中增加了一个成员变量time_slice，用来记录进程当前的可运行时间片段。这是由于RR调度算法需要考虑执行进程的运行时间不能太长。在每个timer到时的时候，操作系统会递减当前执行进程的time_slice，当time_silce为0时，就意味着这个进程运行了一段时间（这个时间片段称为进程的时间片），需要把CPU让给其他进程执行，于是操作系统就需要让此进程重新回到rq的队列尾，且重置

此进程的时间片为就绪队列的成员变量最大时间片max_time_slice值，然后再从rq的队列头取出一个新的进程执行。下面来分析一下其调度算法的实现。RR_dequeue和RR_pick_next的函数实现与FCFS_dequeue和FCFS_pick_next函数实现一致，这里不再赘述。

RR_enqueue的函数实现如下：

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

即把某进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为rq成员变量max_time_slice。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。

RR_proc_tick的函数实现如下：

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

即每次timer到时后，trap函数将会间接调用此函数来把当前执行进程的时间片time_slice减一。如果time_slice降到零，则设置此进程成员变量need_resched标识为1，这样在下一次中断来后执行trap函数时，会由于当前进程程成员变量need_resched标识为1而执行schedule函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。

MLFQ调度算法的实现

当前MLFQ调度算法其实是对RR算法的进一步扩展，即把一个rq队列扩展为了n个rq队列。多个rq队列确保了长时间运行的进程优先级会随着执行时间的增加而降低，而短时间运行的进程会优于长时间运行的进程被先调度执行。在proj13.2中实现了一个比较简单的MLFQ调度算法，其就绪队列由4个双向链表组成（在sched.c中定义）：

```
static struct run_queue __rq[4];
```

第i个rq的最大时间片为8*（1<<i），其中0<=i<4。创建的新进程首先会放到第0个rq中，这样新创建的进程的时间片为8（即第0个rq的最大时间片），如果进程用完了其时间片，则会降到第1个rq中，此时这个进程的时间片设置为16 （即第1个rq的最大时间片），持续下去，直到第3个rq，此时，此时这个进程的时间片设置为64 （即第3个rq的最大时间片）。

下面来分析一下其调度算法的实现。MLFQ_dequeue和MLFQ_proc_tick的函数实现就是直接调用RR_dequeue和RR_proc_tick函数，这里不再赘述。

MLFQ_enqueue的函数实现如下：

```
static void
MLFQ_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    struct run_queue *nrq = rq;
    if (proc->rq != NULL && proc->time_slice == 0) {
        nrq = le2rq(list_next(&(proc->rq->rq_link)), rq_link);
        if (nrq == rq) {
            nrq = proc->rq;
        }
    }
    sched_class->enqueue(nrq, proc);
}
```

如果判断进程proc的rq不为空且time_silce为0,则需要降rq队列，从rq[i]调整到rq[i+1]，如果是最后一个rq，即rq[3]，则继续保持在rq[3]中，然后调用RR_enqueue把proc插入到rq[i+1]中。

MLFQ_pick_next的函数实现如下：

```
static struct proc_struct *
MLFQ_pick_next(struct run_queue *rq) {
    struct proc_struct *next;
    list_entry_t *list = &(rq->rq_link), *le = list;
    do {
        if ((next = sched_class->pick_next(le2rq(le, rq_link))) != NULL) {
            break;
        }
        le = list_next(le);
    } while (le != list);
    return next;
}
```

按顺序从rq[0]~rq[3]依次搜寻，直接调用RR_pick_next来查找某rq链表的头指向的就绪进程，只要找到就返回。

目前的实现其实相对简化了不少。我们把睡眠时间长的进程设定为IO-bounded进程。如果有一个进程经常等待某IO 事件（比如鼠标移动或按键），然后占用一部分CPU时间，但整体上执行了很长时间，则这类进程也算是IO-bounded进程，需要得到及时响应。如果采用上述MLFQ调度算法，则此进程也会落到rq[3]中执行，导致这类进程无法及时响应IO事件。出现这种原因的情况是上述调度算法只考虑了对运行的进程做降级惩罚操作（即只是根据运行时间的增加来递增i，使得运行时间长的进程最终都会落到rq[3]中），但对于睡眠很久的进程没有做升级奖励操作（即没有根据睡眠时间的增加来递减i，使得睡眠时间长的进程能够尽量保持在rq[0]中）。

【问题】对于新创建的子进程，在FCFS/RR/MLFQ调度算法如何设定其时间片比较合理？

【问题】如何扩展MLFQ调度算法，可减少长期运行的IO-bounded进程的响应时间？

【问题】如果需要实现内核级抢占（kernel preemptive）,需要如何设计ucore？

附录

ucore历史

写一个教学OS的初衷是陈渝老师和向勇老师想参考MIT的xv6/JOS开发一个能够与OS课程教材相配套的OS实验环境。没有直接采用xv6/JOS的原因是当时（2008年）xv6没有完整的保护模式页机制和虚存管理机制，JOS不是传统的UNIX 单体内核架构，而是Exokerne内核架构，与当前OS教学的知识点有点远，在互联网上找了一圈，没有合适的。有人说为何不用Linux？其实Linux确实挺好的，只是对于首次学习OS原理的本科生要在短短一学期内搞懂Linux的部分实现细节，可能付出的代价会比较大，需要冒着挂掉其他课的风险。为此陈渝老师鼓励他带的硕士研究生王乃峥试试能否仿照xv6和linux自己鼓捣一个教学用的小OS，并用Ken Thompson和Linus在短短3个月分别开发了UNIX和Linux的故事来从精神上激励他。王乃峥同学看了xv6的代码，本着试试看的想法，就开始coding，并查看各种相关文档和资料，发现也只花了短短一个月不到的时间就完成了支持lab1实验的ucore OS；为此信心大增，以月为单位又接连完成了支持lab2~lab8的ucore OS，前后大约花了8个月（这8个月还顺便完成了减轻体重和找女朋友的重要工作）。做完此事后，王乃峥同学离毕业只有3个时间了。有了之前OS开发的底子，他在3个月的时间内，完成了Linux kernel相关的硕士课题，顺利毕业，开始了他的创业生涯。

在此之后，茅俊杰同学接手了ucore的进一步改进，在他直博期间长期担任操作系统课程的助教，他做的一些有意思包括，扩展ucore-plus（ucore加强版）支持用户态ucore的设计实现（课程大实验），移植到mips/openrisc等CPU上等。目前已经顺利毕业，开始了在产业界的工作。再后来，又有不少同学进行了尝试，比如最近（2017年）张蔚同学把ucore移植到了开源的RISC-V32 CPU上，并准备在2018年做助教辅导他的同班同学。

而向勇老师和陈渝老师鼓励和引导后续的学生继续着操作系统教学和科研的快乐之旅。茅俊杰、陈宇恒、刘聪、杨扬、渠准、任胜伟、朱文雷、曹正、沈彤、陈旭、蓝昶、方宇剑、韩文涛、张凯成、郭晓林、薛天凡、胡刚、刘超、粟裕、袁昕颢、张宇翔、王邈、张蔚、雷凯翔等同学在他们学习OS课程和实践ucore OS的过程中给老师们留下了深刻的印象。目前发现ucore中有不少的bug（不过少于Linux的bug），陈渝老师准备带着学生再研究一些算法、方法和工具，能够在ucore运行前通过静态分析的方法发现其潜在的bug，而且希望能够在ucore在编译时或崩溃时，找到内核代码的bug在哪里，并能分析出为何这个内核代码会导致ucore崩溃的因果链（如果有人感兴趣做这方面的尝试，欢迎直接电邮/电话陈渝老师！）。希望这样能够减轻大家学习OS实验的负担。

ucore lab中的工程项目列表

1. lab1 ：bootloader启动操作系统
2. lab2 ：物理内存管理
3. lab3 ：虚拟内存管理
4. lab4 ：内核线程
5. lab5 ：用户进程
6. lab6 ：处理器调度
7. lab7 : 同步互斥和进程间通信（IPC）
8. lab8 : 文件系统

---

1. lab1 ：bootloader启动操作系统

启动/保护模式

- proj1 : bootloader 能切换到x86-32保护模式且能够通过串口、并口、显示器来显示字符串
- proj2 (<--proj1) : bootloader能读磁盘且分析加载ELF格式的文件
- proj3 (<--proj2) : bootloader能ELF执行文件格式的 ucore toy OS，目前这个toy OS只能答应字符串

显示函数调用栈

- proj3.1 (<--proj3) : ucore能输出函数调用栈信息(包括函数名和行号)，这样便于OS出错后分析问题

响应外设中断

- proj4 (<--proj3.1) : ucore可处理从串口(COM1)、键盘、时钟外设来的中断

支持用户态和内核态，以及系统调用机制

- proj4.1 (<--proj4) : 为了支持proj 4.1.1/2系统调用机制，ucore重新初始化并增加了用户态的代码段和数据段
- proj4.1.1(<--proj4.1) : 用x86的中断机制实现系统调用机制
- proj4.1.2(<--proj4.1) : 用x86的门（gate）机制实现系统调用机制

支持远程gdb调试 (附加部分，其实通过qemu的gdb remote server也可实现大部分功能)

- proj4.2 (<--proj4.1) : ucore增加 gdb remote server/stub，这样可以通过gdb远程调试ucore
- proj4.3 (<--proj4.2) : ucore支持硬件breakpoint和watchpoint，从而具有内部debugger功能

2. lab2 ：物理内存管理

物理内存管理

- proj5 (<--proj4.3) : ucore支持保护模式下的分页机制，并能够管理物理内存

OS教材上的连续物理内存分配算法

- proj5.1 (<--proj5) : 最佳适配算法
- proj5.1.1 (<--proj5.1) : 首次适配算法
- proj5.1.2 (<--proj5.1) : 最坏适配算法

实际OS中的以页大小（4KB）为单位的连续物理内存分配算法

- proj5.2 (<--proj5.1) : 伙伴（buddy）分配算法

实际OS中的小于页大小的连续物理内存分配算法

- proj6 (<--proj5.2) : SLAB内存分配算法

3. lab3 ：虚拟内存管理

支持页访问错误异常的处理

- proj7 (<--proj6) : 能够有页访问错误异常的处理机制，提供了虚存管理（VMM）的框架

提供swap机制，为能够实现各种页替换算法做好准备

- proj8 (<--proj7) : 实现swap in/out机制，并加入页替换算法的实现框架

实现map/unmap机制，并提供dup, exit等函数，为后续进程管理（比如创建子进程等）做好准备

- proj9 (<--proj8) : 增加内核函数map, unmap, dup, exit等

实现share memory机制，为后续进程间共享内存空间做好准备

- proj9.1 (<--proj9) : 实现 shmem_t内存结构，并完成香港函数，实现share memory

3 实现COW机制，为高效创建子进程做好准备

- proj9.2 (<--proj9.1) : 实现支持高效进程复制的虚存核心功能Copy On Write（简称COW）

4. lab4 ：内核线程

创建内核线程，此时需要引入调度，进程上下文切换等机制

- proj10 (<--proj9.2) : 实现线程和进程管理的关键数据结构进程控制块（Process Control Block, 简称PCB），完成对内核线程的创建所需功能，并建立基本的调度机制，主要是体现能够切换两个内核线程。

5. lab5 ：用户进程

进程管理框架

- proj10.1 (<--proj10) : 实现用户进程管理框架，并完成与创建用户进程相关的内核函数（读ELF格式的文件、fork、execve），以及与调度进程相关的调度器

与进程生命周期相关的系统调用

- proj10.2 (<--proj10.1) : 实现进程管理相关的系统调用 wait、kill、exit

进程中的堆管理系统调用sys_brk

- proj10.3 (<--proj10.2) : 完成管理用户进程的内存堆（heap）的系统调用sys_brk

与进程生命周期相关的涉及睡眠和唤醒的系统调用

- proj10.4 (<--proj10.3) : 完成用户进程调度相关的函数sleep，并增加timer的功能支持

采用内核线程机制，能够根据系统状态更好地动态支持swap in/out虚存功能

- proj11 (<--proj10.4) : 用内核线程方式实现虚存的swap机制

实现用户态的线程（基本原理与Linux的轻量级进程一致）

- proj12 (<--proj11) : 实现系统调用map、unmap和共享内存share memory，实现用户态线程机制;

6. lab6 ：处理器调度

OS教材上的调度算法

- proj13 (<--proj12) : 实现通用调度框架和简单的想来先服务（First Come First Serve，简称FCFS）调度算法
- proj13.1 (<--proj13) : 实现轮转（RoundRobin，简称RR）掉短算法
- proj13.2 (<--proj13.1) : 实现多级反馈队列（MultiLevel Feedback Queue，简称MLFQ）调度算法

7. lab7 : 同步互斥和进程间通信（IPC）

OS教材上的信号量机制

- proj14 (<--proj13.2) : 实现内核中的信号量（semaphore）机制

用户态进程的信号量，需要考虑一些实际情况（比如一个获得信号量的进程湍溃了）

- proj14.1 (<--proj14) : 实现用于用户态进程/线程的信号量机制,
- proj14.2 (<--proj14.1) : 增加在信号量等待中的超时判断机制

其他一些IPC机制，在一些实时OS中常见

- proj15 (<--proj14.2) : 实现事件（event ）IPC机制
- proj16 (<--proj15) : 实现邮箱（mailbox）IPC机制

OS教材上的管程（Monitor）和条件变量机制

- proj16.1 (<--proj16) : 实现管程和条件变量

8. lab8 : 文件系统

建立虚拟文件系统，把设备按文件來管理

- proj17 (<--proj16) : 实现vfs框架, file数据结构和相关操作，文件化各种输入输出外设 (stdin, stdout, null)

按照文件的方式实现一种UNIX中常见的IPC机制--管道（PIPE）

- proj17.1 (<--proj17) : 实现匿名管道（PIPE）和有名管道（FIFO）

增加SFS （简单文件系统--sfs，并实现用户进程访问文件所涉及的函数）

- proj18 (<--proj17.1) : 在VFS上增加具体文件系统实例sfs 'simple filesystem'和对应的文件操作相关函数

增加sfs中目录访问相关的函数

- proj18.1 (<--proj18) : 增加mkdir/link/rename/unlink (hard link)相关的系统调用和内核函数

实现了通过exec加载存储在磁盘上的执行文件并创建/执行进程的功能

实现exec功能

- proj18.2 (<--proj18) : add exec

扩展exec功能，在加载运行应用程序能够带上执行参数

- proj18.3 (<--proj18.2) : add exec with arguments (at most 32)

在上述ucore OS的支持上实现了shell（一个用户态的命令行交互执行程序）

- proj19 (<--proj18.3) : shell

开发维护人员

- 当前维护者
  - 陈渝 http://soft.cs.tsinghua.edu.cn/~chen yuchen@tsinghua.edu.cn
  - 茅俊杰 eternal.n08@gmail.com
  - 向勇 xyong@tsinghua.edu.cn
- 贡献者

  茅俊杰、陈宇恒、刘聪、杨扬、渠准、任胜伟、朱文雷、 曹正、沈彤、陈旭、蓝昶、方宇剑、韩文涛、张凯成、 S郭晓林、薛天凡、胡刚、刘超、粟裕、袁昕颢...

ucore实验中的常用工具

在ucore实验中，一些基本的常用工具如下：

- 命令行shell: bash shell -- 有对文件和目录操作的各种命令，如ls、cd、rm、pwd...
- 系统维护工具：apt、git
  - apt：安装管理各种软件，主要在debian, ubuntu linux系统中
  - git：开发软件的版本维护工具
- 源码阅读与编辑工具：eclipse-CDT、understand、gedit、vim
  - Eclipse-CDT：基于Eclipse的C/C++集成开发环境、跨平台、丰富的分析理解代码的功能，可与qemu结合，联机源码级Debug uCore OS。
  - Understand：商业软件、跨平台、丰富的分析理解代码的功能，Windows上有类似的sourceinsight软件
  - gedit：Linux中的常用文本编辑，Windows上有类似的notepad
  - vim: Linux/unix中的传统编辑器，类似有emacs等，可通过exuberant-ctags、cscope等实现代码定位
- 源码比较工具：diff、meld，用于比较不同目录或不同文件的区别
  - diff是命令行工具，使用简单
  - meld是图形界面的工具，功能相对直观和方便，类似的工具还有 kdiff3、diffmerge、P4merge
- 开发编译调试工具：gcc 、gdb 、make
  - gcc：C语言编译器
  - gdb：执行程序调试器
  - ld：链接器
  - objdump：对E执行程序文件进行反编译、转换执行格式等操作的工具
  - nm：查看执行文件中的变量、函数的地址
  - readelf：分析ELF格式的执行程序文件
  - make：软件工程管理工具， make命令执行时，需要一个 makefile 文件，以告诉make命令如何去编译和链接程序
  - dd：读写数据到文件和设备中的工具
- 硬件模拟器：qemu -- qemu可模拟多种CPU硬件环境，本实验中，用于模拟一台 intel x86-32的计算机系统。类似的工具还有BOCHS, SkyEye等

上述工具的使用方法在线信息

- apt-get
  - http://wiki.ubuntu.org.cn/Apt-

get%E4%BD%BF%E7%94%A8%E6%8C%87%E5%8D%97

- git
  - http://www.cnblogs.com/cspku/articles/Git_cmds.html
- gcc
  - http://wiki.ubuntu.org.cn/Gcchowto
  - http://wiki.ubuntu.org.cn/Compiling_Cpp
  - http://wiki.ubuntu.org.cn/C_Cpp_IDE
  - http://wiki.ubuntu.org.cn/C%E8%AF%AD%E8%A8%80%E7%AE%80%E8%A6%81%E8%AF%AD%E6%B3%95%E6%8C%87%E5%8D%97
- gdb
  - http://wiki.ubuntu.org.cn/%E7%94%A8GDB%E8%B0%83%E8%AF%95%E7%A8%8B%E5%BA%8F
- make & makefile
  - http://wiki.ubuntu.com.cn/index.php?title=%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile&variant=zh-cn
  - http://blog.csdn.net/a_ran/article/details/43937041
- shell
  - http://wiki.ubuntu.org.cn/Shell%E7%BC%96%E7%A8%8B%E5%9F%BA%E7%A1%80
  - http://wiki.ubuntu.org.cn/%E9%AB%98%E7%BA%A7Bash%E8%84%9A%E6%9C%AC%E7%BC%96%E7%A8%8B%E6%8C%87%E5%8D%97
- understand
  - http://blog.csdn.net/qwang24/article/details/4064975
- vim
  - http://www.httpy.com/html/wangluobiancheng/Perljiaocheng/2014/0613/93894.html
  - http://wenku.baidu.com/view/4b004dd5360cba1aa811da77.html
- meld
  - https://linuxtoy.org/archives/meld-2.html
- qemu
  - http://wenku.baidu.com/view/04c0116aa45177232f60a2eb.html
- Eclipse-CDT
  - http://blog.csdn.net/anzhu_111/article/details/5946634

MOOC OS 相关资料

OS基本概念和原理

- MOOC OS 2015 on 学堂在线 https://www.xuetangx.com/courses/TsinghuaX/30240243X/2015_T1/about
- MOOC OS 2014 on TOPU http://www.topu.com/mooc/4100

OS设计与实现细节

- OS实验代码 https://github.com/chyyuu/mooc_os_lab

动手实践OS

- "操作系统简单实现与基本原理 — 基于ucore" http://chyyuu.gitbooks.io/ucorebook/
- "操作系统简单实现与基本原理 — 基于ucore" 配套代码 https://github.com/chyyuu/ucorebook_code
- ucore plus 跨硬件平台的ucore OS https://github.com/chyyuu/ucore_plus

MOOC OS 2015 WIKI

- http://os.cs.tsinghua.edu.cn/oscourse/OS2015

在线交流

- 清华计算机系MOOC OS课程在线QA平台
- QQ群 181873534 主要用于事件通知，聊天等

课程汇总信息

- 课程汇总

版权信息

ucore OS是用于清华大学计算机系本科操作系统课程的OS教学试验内容。 ucore OS起源于MIT CSAIL PDOS课题组开发的xv6&jos、哈佛大学开发的 OS161教学操作系统、以及Linux-2.4内核。